DIRECTION DE LA RECHERCHE TECHNOLOGIQUE

CEA/SACLAY

DEPARTEMENT INGENIERIE
LOGICIELS ET SYSTEMES

Fluctuat - User manual

Static analysis of numerical programs.

DIRECTION DE LA RECHERCHE TECHNOLOGIQUE
LABORATOIRE D'INTEGRATION DES SYSTEMES ET DES TECHNOLOGIES
DEPARTEMENT INGENIERIE LOGICIELS ET SYSTEMES
CEA/SACLAY - 91191 GIF-SUR-YVETTE CEDEX
TEL 01 69 08 55 32 - FAX 01 69 08 83 95 - E-MAIL fabrice.derepas@cea.fr

Etablissement public à caractère industriel et commercial

R.C.S PARIS B 775 685 019

Table des matières

1 Introduction and principle **4**
- 1.1 Finite precision and IEEE 754 norm . . . 4
- 1.2 Principle of Fluctuat . . . 4

2 Using Fluctuat **5**
- 2.1 Analysis phases . . . 5
- 2.2 GUI (main window) . . . 5
- 2.3 Project management . . . 6
  - 2.3.1 Project creation . . . 6
  - 2.3.2 Open an already defined project . . . 7
  - 2.3.3 Edition of the source files . . . 8
- 2.4 Analysis phase . . . 8
  - 2.4.1 Starting the analysis . . . 8
  - 2.4.2 The “Simple parameters” mode . . . 9
  - 2.4.3 The “Advanced parameters” mode . . . 11
  - 2.4.4 Monitor . . . 19
- 2.5 Visualisation of the results . . . 19
  - 2.5.1 Analysis information . . . 19
  - 2.5.2 Variable values and errors . . . 20
  - 2.5.3 Evolution graphs . . . 21
  - 2.5.4 Warnings . . . 21
  - 2.5.5 In/out variables . . . 22
  - 2.5.6 Dead code . . . 22
  - 2.5.7 Save results . . . 23
  - 2.5.8 Draw results (2D/3D) . . . 23
  - 2.5.9 Show Assertions . . . 24
  - 2.5.10 Worst-case generation . . . 25
  - 2.5.11 Sensitivity analysis . . . 26
- 2.6 Batch mode . . . 26
  - 2.6.1 Command-line scripts . . . 27
  - 2.6.2 Input and output files . . . 27
- 2.7 Interactive static analysis . . . 30
  - 2.7.1 Invoking the interactive mode . . . 31
  - 2.7.2 commands of the interactive mode . . . 32
- 2.8 License . . . 34

3 Assertions in the source code **35**
- 3.1 Specifying ranges on values of variables . . . 35
- 3.2 Specifying ranges on values and errors of variables . . . 36
- 3.3 Limiting ranges of values and errors of variables . . . 38
- 3.4 Proof assertions . . . 39
- 3.5 Assertions on the dynamics of the inputs of a system . . . 40
- 3.6 Join and meet assertions . . . 40
- 3.7 Assertions to print information during the analysis . . . 41
- 3.8 Assertions to retrieve input-output relations . . . 41
- 3.9 Local and Manual Subdivisions . . . 42
  - 3.9.1 Regular Subdivisions . . . 42
  - 3.9.2 Irregular Subdivisions . . . 43
  - 3.9.3 Subdivisions to Meet a Requirement . . . 44
  - 3.9.4 Manual Subdivisions . . . 45

4 Examples **47**
4.1 Simple examples . . . 47
4.2 Use of assertions . . . 48
4.2.1 Helping the analyser . . . 48
4.2.2 Creating stubs . . . 48
4.2.3 Specifying constants . . . 49
4.2.4 Limit values and errors . . . 49
4.3 Play with parameters . . . 50
4.3.1 Fixpoint computations in loops . . . 50
4.3.2 Call stack depth . . . 57
4.3.3 Subdivide bitwise operations . . . 57
4.4 Use of C ANSI features . . . 58
4.4.1 Integers . . . 58
4.4.2 Complex data structures . . . 59
4.4.3 Unspecified behaviours . . . 61
4.5 Warnings . . . 63
4.5.1 Invalid operations . . . 63
4.5.2 Invalid conversions . . . 63
4.5.3 Programs that never end . . . 63
4.5.4 Unstable tests . . . 64
4.5.5 Fluctuat specific warnings . . . 65

5 Inside Fluctuat **67**
5.1 Introduction . . . 67
5.2 Static analysis . . . 67
5.2.1 Interval arithmetic . . . 67
5.2.2 Affine arithmetic . . . 67
5.2.3 Iteration algorithm . . . 68
5.3 Semantics . . . 70
5.3.1 Semantic of floating-point numbers . . . 70
5.3.2 Semantic of fixed-point numbers . . . 72
5.4 Domains . . . 72
5.5 Implementation of the analyser . . . 72
5.5.1 Implemented operations . . . 72
5.5.2 Some specificities of the implementation . . . 73
5.5.3 Known limitations and bugs . . . 74

A Installation of the tool **75**

B Actions reference card **77**

C Interactive analysis reference card **78**

D Simple to advanced parameters **79**

E Parameters **80**

1 Introduction and principle

1.1 Finite precision and IEEE 754 norm

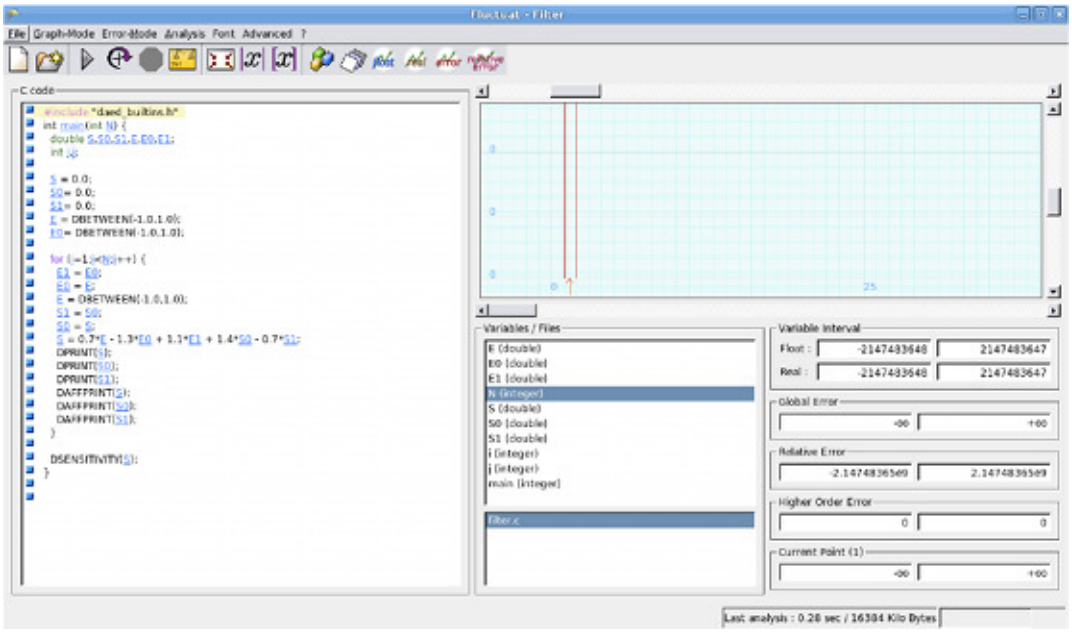
The manipulation of real numbers by computers is approximated by floating-point arithmetic, which uses a finite representation of numbers. This implies that a (small in general) rounding error may be committed at each operation. Although this approximation is accurate enough for most applications, there are some cases where results become irrelevant because of the precision lost at some stages of the computation, even when the underlying numerical scheme is stable.

The IEEE-754 standard[3] defines the format of floating point numbers and how the floating point computations are exactly rounded on every processor supporting it.

1.2 Principle of Fluctuat

This document presents Fluctuat, which is an abstract-interpretation based static analysis tool for characterizing (in a guaranteed manner) the propagation of rounding errors in floating-point computations, for sets of possible executions. Its aim is to detect automatically a possible catastrophic loss of precision, and its source. It allows to find possible programming errors such as potentially dangerous double/float conversions, or blatant instabilities or losses of accuracy. The tool is intended to cope with real industrial problems, and we believe it is especially appropriate for critical instrumentation software. Its output is in particular for each scalar variable, and at the last control point of the function which is analysed :

- an interval, super-set of possible floating-point values,
- an imprecision error, with respect to the semantics involving real numbers, decomposed by contribution of each line of source code,
- an interval, super-set of the higher-order imprecision (global) error with respect to the "real" semantics.



It can also be used to find :

- propagation of uncertaineties (inputs, parameters),

– invariants (in particular ranges of variables) on the float and real number semantics,
– run-time execution errors (such as... ?),
– functional proofs both for the program (floating point semantic) and for the underlying algorithm (real number semantic),
– test case generation (for maximal (minimal) value or error),
– sensitivity analysis.

2 Using Fluctuat

2.1 Analysis phases

There are different phases in the use of the tool. For a given problem the user has to go through three phases : project creation, static analysis and visualisation of the results.

A project is constituted of :
– one or several sources files (`.c` or `.ada`),
– one or several include directories (equivalent to the `-I` gcc option),
– some other compiling options (such as `-D`).
– the name of the function where the analysis starts,

This is stored internally in a project file called `project.xml`, which contains all the previous information.

Once the project is built, the analysis phase can start. To perform the analysis, Fluctuat uses a project, and optionally a set of parameters.
With these parameters, you can choose the behavior of the analyzer, from the precision of its computations, to the rounding mode of your environment.
And after the analysis, Fluctuat writes all the information in the `output` directory. In this directory, you can find information about values and errors of each variable. This is where the GUI retrieves the information. Each of these phases are detailed in the following sections.

Fluctuat can be used in graphical or in batch mode. The major part of the operations done in the graphical mode can also be achieved in batch mode. The following sections show how to use Fluctuat in graphical mode, but Section 2.6 explains how to do the same operations in batch.

2.2 GUI (main window)

The main window (Figure 1) is divided in several parts :
– `Menus`, you can find all the actions of the interface in this part and some quick links,
– `Source code`, this part display the contents of the current file,
– `Origins`, each bar represents a part of the current imprecision error,
– `Variables`, all the variables readable at the end of the program are displayed here,
– `Files`, to navigate between source files,
– `Values and errors`, the information of the selected variable,
– `Status bar`, to find useful information about the analysis.

All the actions (and the quick link corresponding) of the `Menus` bar are detailed in Appendix B.

The `Source code` part displays the contents of the current file. You can change the font used to display code, see actions `Small font` and `Big font`.

Once the first analysis is done, variables of the program become underlined, and you can access a variable (values and errors) by clicking on it on the source code, or by selecting it in the `Variables` part. You can also change the current file displayed with the `Files` part.

If a variable is selected, the `Values and errors` and `Origins` parts display information collected by the tool during the last analysis.
The `Origins` part is a representation of the error, detailed by each contribution of each line of the program

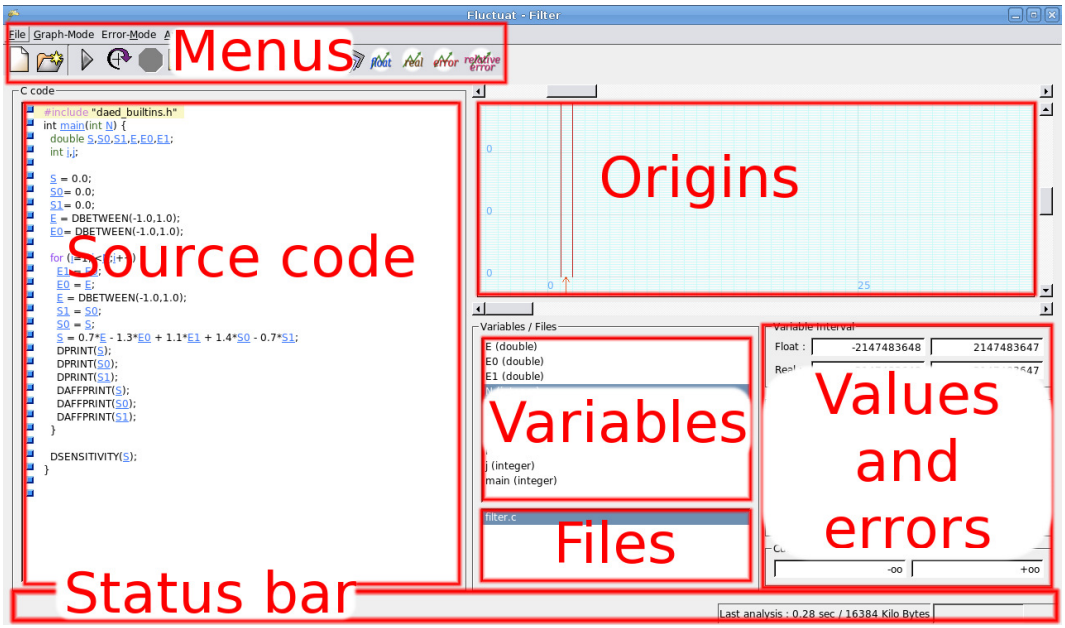


FIGURE 1 – Main window

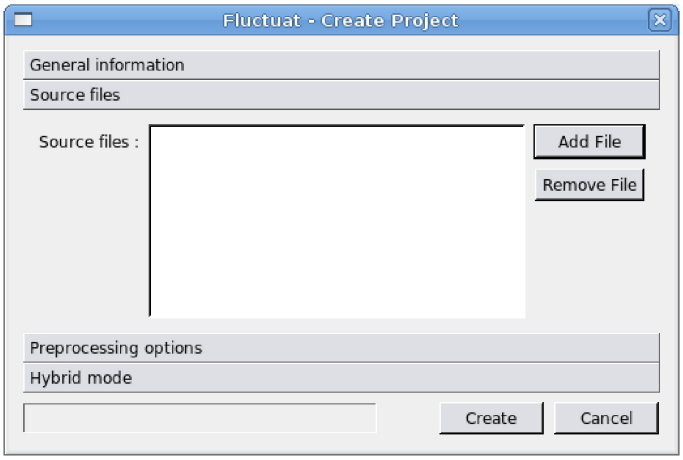
(see Section 2.5.2). The `Variable interval` section gives two intervals. The first contains all the possible floating-point values the variable can take at the end of the program, and the second is for the real values. The block below indicates the possible values of error, relative error and higher-order error (see Section 2.5.2 for details).

The `Status bar` indicates information about the last analysis. In the left part, a status message ("Analysis terminated", ...) can be displayed. In the right part, the statistics about the last analysis are displayed, and a small monitor is able to show the progress of the running analysis (see Section 2.4.4 for more information on monitor).

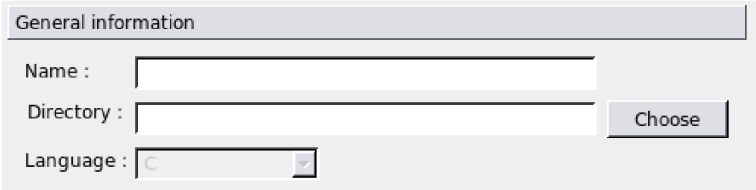
2.3 Project management

2.3.1 Project creation

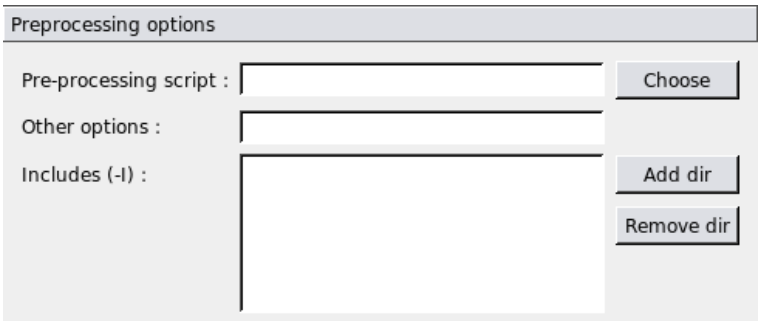
When selecting `Create Project`, the following window opens :



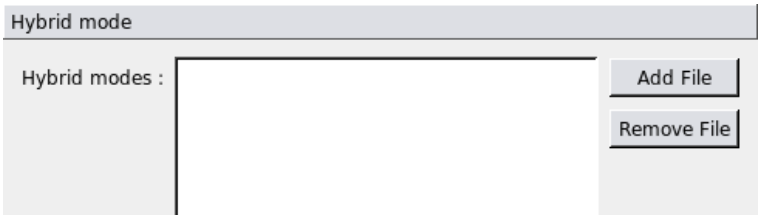
First, you have to select one or more source files to be analyzed. Even if Fluctuat manages C and Ada programs, it is impossible to mix C and Ada files. Once source files are selected, the `General information` item is automatically filled. At this point, you can still change the name of the project as well as the directory in which it will be stored. The default project name is determined by the name of the first source file of the project, and similarly ? ? the directory.



The next item is `Pre-processing options`. You can specify a `pre-processing script`. The script must be written in `Bash` or `Sh` language. It takes in input the name of the resulting file, following by the name of input files. Some general scripts are provided with Fluctuat. In this section, you can also specify compiling options like gcc option `-D`, and directories containing the header files .h needed by your source files.



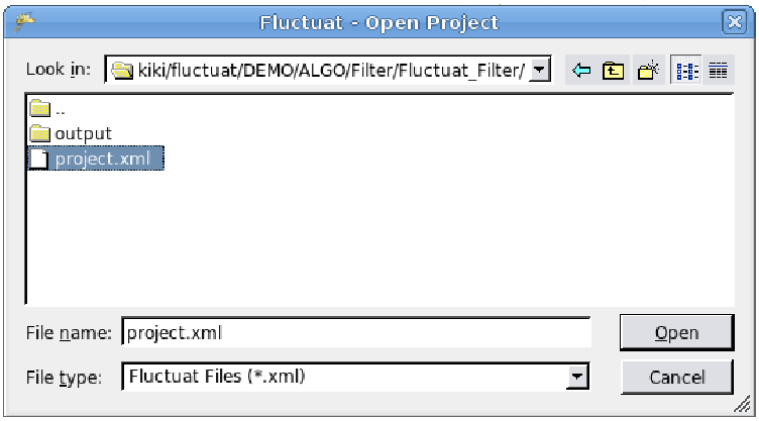
In the case that Fluctuat is provided with the GRKLib integrator, you can specify in the last section (`Hybrid mode`) some header files, specifying Ode's used in the analyzed program. For more information, we refer the reader to GRKLib documentation.



Once you finish the selection of source files and options, you can click on the `Create` button to start the project creation. This process creates project files, pre-processes source code, and stops in case of a compilation error.
If the project creation does encounter a problem, then the compilation errors can be seen in the error log. Otherwise, the analyse phase can begin.

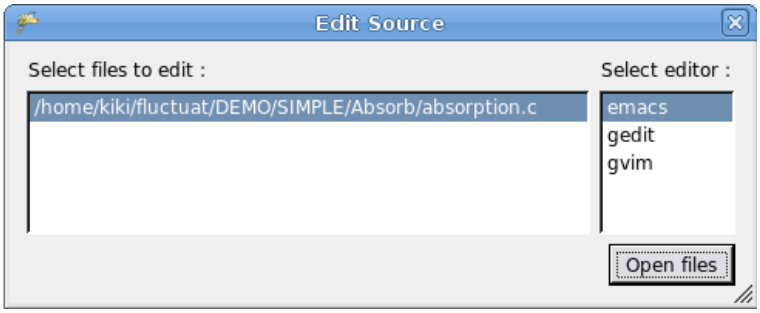
2.3.2 Open an already defined project

To perform an analysis, or simply to view old results, you can use the action `Open Project`. It opens the following window in which you have to select a `project.xml` file. If the project was created by Fluctuat, this file is located inside the folder "Fluctuat_<ProjectName>".

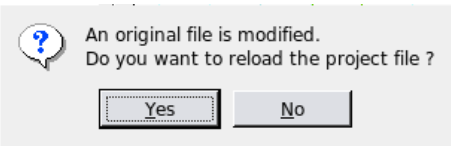


2.3.3 Edition of the source files

To modify one or all the source files of your project, you can use the `Edit Source` action. It opens the following window, and you have to select files you want to edit, the editor, and then click on `Edit files` button.



When the modifications are saved, Fluctuat detects that the project is modified and asks you if you want to reload the file :

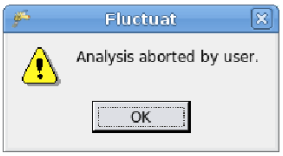


When you check yes Fluctuat restarts the creation phase.

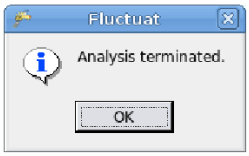
2.4 Analysis phase

2.4.1 Starting the analysis

The next step is to perform the analysis. It is done through buttons `Start Analysis` and `Quick Analysis`. The difference between these actions is the selection of parameters. With the latter, this selection is skipped. It is not the case with the former. Although Fluctuat has default parameters, the `Quick Analysis` button can be used only if at least one analysis was run before. It then reuses the saved parameters. If you want to interrupt an analysis, use the `Abort Analysis` button. The analysis then ends on windows :



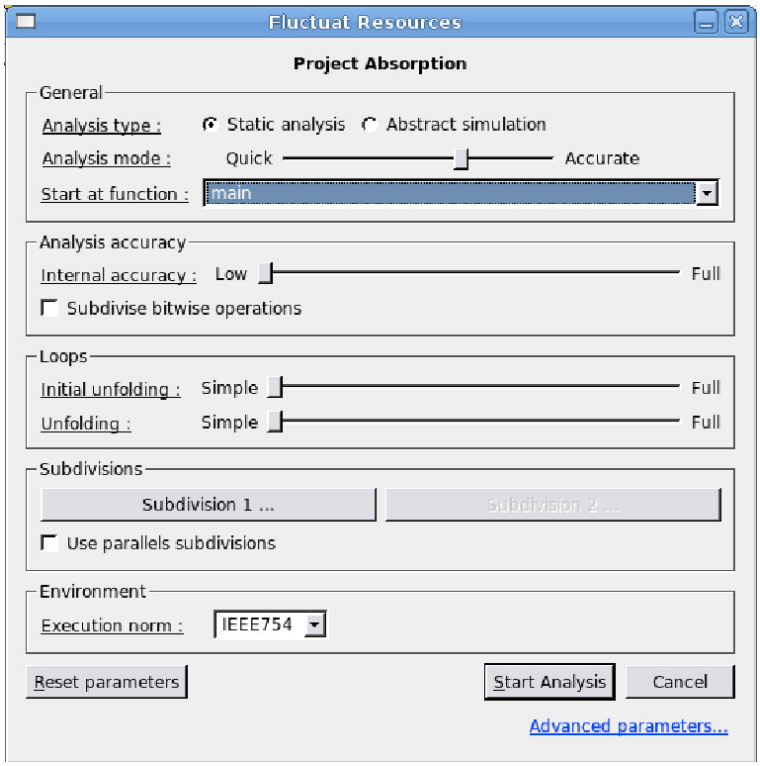
Otherwise, you have to wait for the end of the analysis :



When you click on the `Start Analysis` button, the parameters selection window appears. Section 2.4.2 explains how to select these parameters. If you want to modify more parameters, you can click on the button `Advanced Parameters` in the bottom of the parameters window. Section 2.4.3 explains how to select the advanced parameters. The correspondence between the two settings is detailed in Appendix D.

2.4.2 The "Simple parameters" mode

The parameter's window looks like :



We comment below the different items that the user can modify in this mode :

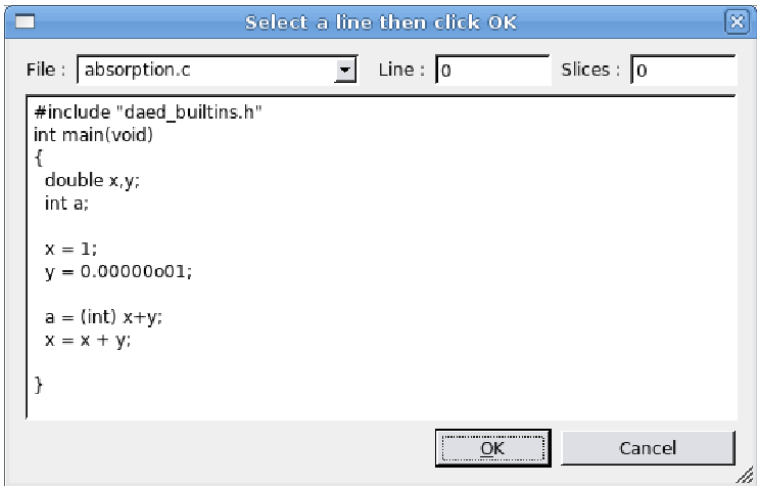
General First, you have to choose the `analysis type` between static analysis and abstract simulation. The `Abstract simulation` mode is an alternative to propagate abstract values in a program, with a unique value, but possibly an error interval. This allows, for an execution, to have a precise estimate of the errors and causes of errors on an execution trace of the program. Input variables must thus have as value an interval with zero width, but can be perturbed by interval errors. If input variables have their value in an interval with non zero width, the center of the interval is taken as value, and a warning message is issued (that can be seen at the log console). In that mode, instead of computing fixpoints by unions over iterations, the analyzer unrolls completely the loops. The `analysis mode` allows you to select the precision of the analysis. This slider can be set to five different values, from the less accurate (but maybe quicker) to the most accurate (but maybe slow) mode. The recommended (and default) value is the fourth mode. This is of course but an indication : less precise analysis are not always quicker.

Then you have to select the `initial function` where the analysis will start. The resulting values of the analysis will correspond to the values at the end of this function.

Analysis accuracy The slider controls the `internal accuracy` of the analyzer (i.e. the number of bits it will use to make internal calculations). To tune the precision of the results, increase this value. If the checkbox `bitwise operations` is checked, the analyzer will subdivide operations ||, &&, << and >> to improve the precision of computations of imprecision errors.

Loops These parameters are useful when the analyzer fails to find a fixed-point in a loop (i.e. returns a $[-\infty; +\infty]$ interval). The first parameters tells Fluctuat to `unfold initial iterations` of the loops to improve the first steps of the analysis. It is also useful to unroll completely loops in some cases (array initialisation for example). The second parameter **''unfolding''** modifies the behavior of Fluctuat when it `unfolds loops` to find a fixed-point. This of course might slow down the analysis, but will lead to better precision.

Subdivisions Another way to improve the precision of the analysis is to subdivide one or two input intervals. This can be useful on variables that are used in non-linear computations. Non-linear computations include polynomial or transcendental computations, but also "if/then/else" statements, where subdivisions can be used. The tool will subdivide the variables in small intervals of the same length, spanning the entire value domain, then analyzes each part, and finally merge the results of each analysis. To add subdivisions to your program, click on the `Subdivision 1` (for one variable) or `Subdivision 2` buttons (if you have already subdivided one variable). This window appears :



You have to select the file in which the value domain of the variable you want to apply subdivision (i.e. a `DBETWEEN` or the like). Then select the line and specify the number of slices you want to make. When this is done, click on `Ok`.
When using subdivisions, the union of results on values and errors for subdivisions already analysed can be seen via the graphical interface along the analysis. Also, the first time the analyser finds a possibly value or error on a variable infinite in one subdivision, an alarm box opens (see Figure 2.5.4).
If you are running Fluctuat on a multi-processor computer, you can increase the analysis speed by selecting the `Use parallel subdivisions` option.

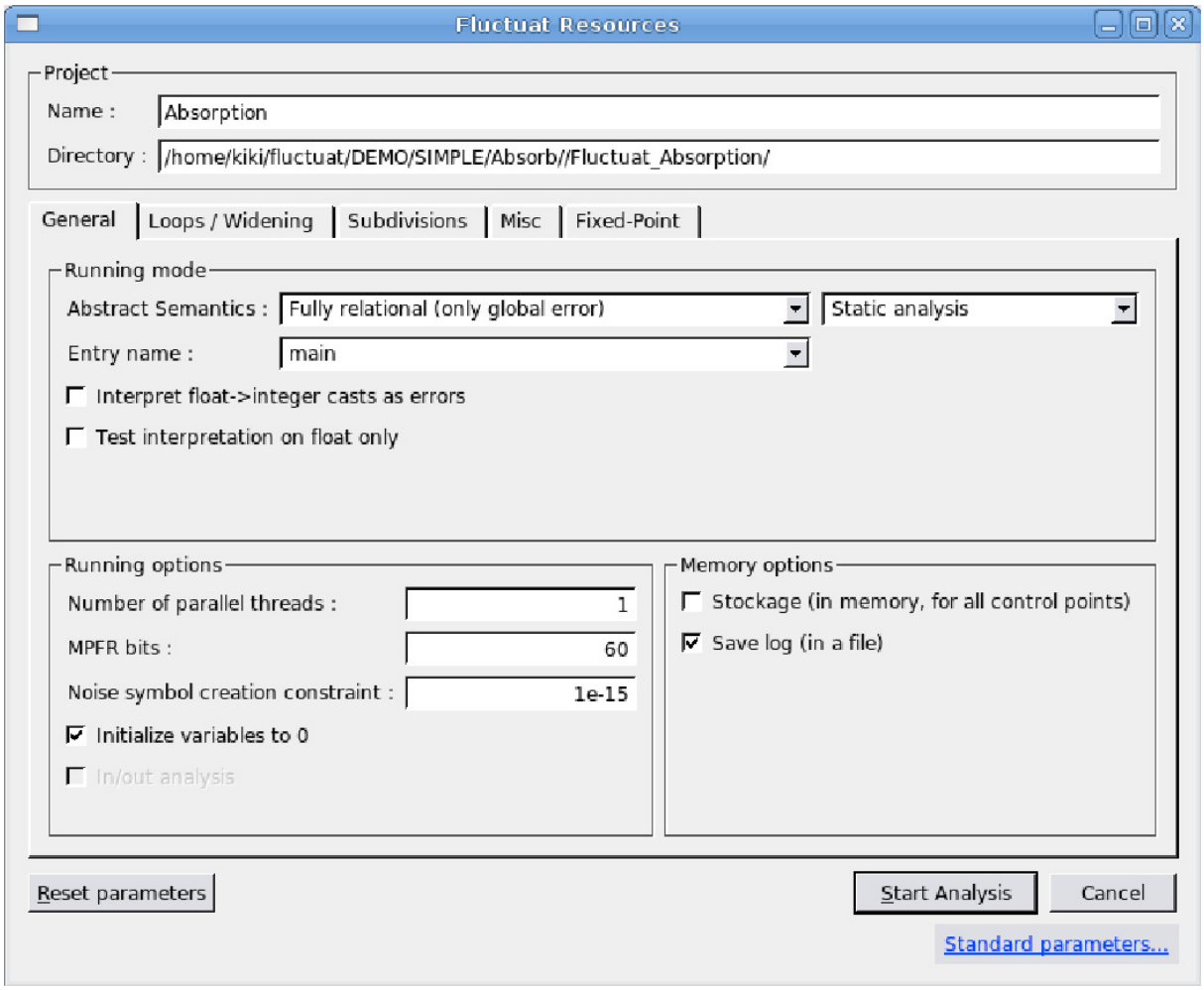
Environment If the program you want to analyse is not compiled on a `IEEE754` compliant processor, you can change the behavior of Fluctuat to specify your processor specifications. The current modes are `IEEE754` and `MIL1750`, but you can also manually specify your architecture behavior in the advanced parameter mode (see Section 2.4.3). `MIL1750` is a standard defined by the U.S. Air Force, and widely used in military

applications (see [2]).
Please contact the Fluctuat development team if you want to add a new mode in this window.

The `Reset parameters` button allows to set all parameters back to their default values.

2.4.3 The "Advanced parameters" mode

This section shows how to select the advanced parameters [1]. This mode is more complex and now we will browse through its sub-menus. Note that the `Reset parameters` button allows to set all parameters (not just the ones defined in the current sub-menu) back to their default values.
We now describe one by one the parameters that can be modified in each tab of the parameters window :



General :

- `Abstract semantics` :
  You can choose between five semantics for the analysis. The first mode, `Non-relational`, derives naturally an abstraction of the concrete semantics described in Section 5.2.1 using intervals. The second mode, `Relational on values`, is the abstraction described in Section 5.2.2. That is relational only on

1. This view becomes the default view if the "FLUCTUATUSER" environment variable is set to "Advanced".

values, and uses the interval abstract domain on error terms. The third mode, `Fully relational`, is the same relational abstraction but for both values and errors. The fourth mode, `Fully relational (only global error)`, is a derivation of the previous mode but the error is less decomposed. This mode is the default mode. It is a good compromise between time cost and precision. And the fifth mode, `Fixed point` selects the mode for the analysis of fixed-point representation of numbers (see Section 5.3.2).

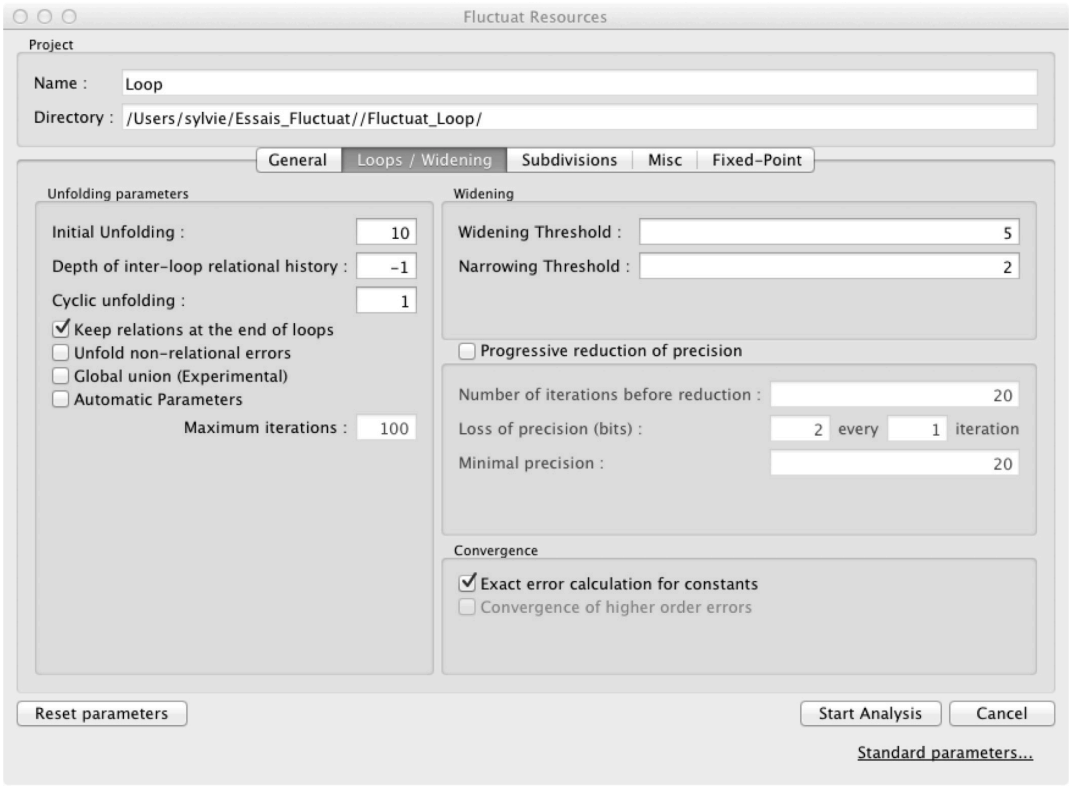
- `Static Analysis / Symbolic Execution` :
  The `Static Analysis` is the default mode. We recall that `Symbolic Execution` (also called `Abstract simulation` is "Simple parameters" mode) is an alternative to propagate abstract values in a program, with a unique (float or real according to ? ?) value, but possibly an error interval. This allows, for an execution, to have a precise estimate of the errors and causes of errors on an execution trace of the program. Input variables must thus have as value an interval with zero width, but can be perturbed by interval errors. If input variables have their value in an interval with non zero width, the center of the interval is taken as value, and a warning message is issued (that can be seen at the log console). In that mode, instead of computing fixpoints by unions over iterations, the analyzer unrolls completely the loops. When this option is chosen, some parameters in the `Loops` and `Widening` sub-menus cannot be set, and the `Automatic Parameters` option cannot be set.
- `Entry Name` :
  The name of the function to analyse in the code (by default, `main`) - of course, all functions called from it are analysed.
- `Interpret float->integer casts as errors` :
  The default interpretation by Fluctuat of cast from a float to an integer is to truncate the fractional part of the values to fit in the integer type, and considering this as the intended behavior. This means no imprecision error is reported. On the contrary, if this option is checked, this cast is interpreted as a loss of precision and an error is generated equal to the fractional part of the value.
- `Test interpretation on float only` :
  Used only with the latest relational mode only. If value is 0 (default value), tests are interpreted on floating-point and real values, and the result is the intersection of the two (relying on the idea that the floating-point and real value control flow are the same, otherwise a warning for an unstable test is issued). If value is 1, tests are interpreted only on floating-point values. With value 0, in the general case more accurate results are obtained than when ticking the box (value 1) in order to interpret tests (and narrow values) only on floating-point numbers. But in some cases this may not give the expected results. See Section **??** for examples of the use of these two modes.
- `Number of parallel threads` :
  This parameter controls an experimental feature, the parallelization of subdivisions. Set to 0 or 1, the feature is disabled. Else you can control the number of threads analysing subdivisions asynchronously.
- `MPFR Bits` :
  Defines the number of bits used for MPFR (internal representation of real numbers used by the analyser). This number should in most cases be greater than the number of bits used for the representation of double precision numbers, that is 53 bits (or 25 bits if only simple precision floating-point variables are used in the C code to analyse). This number should not be greater than 50000. In case the analyzed programs are close to being numerically unstable, it is worth to increase this parameter (see Example 4.3.1). A good way to see if this is necessary is to analyse the program in symbolic execution mode. If the error increases dramatically, it is worth to change this parameter.
- `Noise symbol creation constraint` :
  It is a double value which defines the threshold of error symbol creation. When an error is greater than this threshold, Fluctuat creates a new error symbol. Otherwise, Fluctuat merges smaller errors in a single symbol. To avoid the merge of smaller errors, and then try to obtain better precision of the analysis, you have to set this parameter to 0, at the expense of slowing down the analysis.
- `Initialise variables to 0` :
  Enables to specify if the variables which are non-initialised have value `top` or `0`.

- `In/out analysis` :
  For the time being, this option is possible only with the non relational domain (this latter is automatically selected when the in/out analysis mode is set). The idea is that for large programs, the user may want to make a first fast analysis to determine the in and out status of variables, for example to check that all variables are initialised. And then he can launch a more accurate analysis. Also, the user might want to extract a part of this large program and analyse this component. The user might then want to be sure to know which variables are the input parameters, that he will have to give bounds to (see Section 3).
- `Storage` :
  When the box is not ticked, the abstract states of variables are kept only at control points where they are necessary to the analysis. Otherwise, they are kept at any point of the program (this of course needs more memory space).
- `Save log` :
  Keep this option activated if you want to keep all the information about the analysis went. In case of a software bug in Fluctuat, this file should be sent to the developpers of Fluctua as part of a bug report. Be careful, through the log file can be huge in some cases (with subdivisions for example).

Loops / Widening :

Loops can be either completely unfolded, when a bound to the number of iterations is known, else Fluctuat tries to determine the least set of values taken by variables for every execution of the program by over-approximating the "least fixed point' of the functional abstracting the semantics of the program. In the second case, the analyzer joins values coming from the different iterations for some time before using a widening operator (see Section 5.2.3). The widening operator allows to extrapolate the least fixed point computation towards infinity if the sets of values are still growing after some iterations, this ensures that the analysis terminates in finite time.

You can control the behavior of the computations in loops with the following parameters :



- `Initial Unfolding` :
  Positive or zero integer, that defines the number of time the loop is executed before starting unions over iterations. This initial execution often improves the accuracy of the result by allowing to pass over the often different behaviour of the first iterations of the loop. Also, this parameters allows us to completely unfold loop when a bound to the number of iterations is known.
- `Depth of inter-loop relational history` :
  It is an integer that parametrises the depth of relations wanted between loop iterations. Keeping relations between values at all iterations of loops would be too costly in memory and computing time, but on the contrary keeping none leads to imprecise results in the case of iterative schemes of order greater than one. When this parameter is set to -1 (which is the default value), all relations are kept in the loop. This parameter is completely independent of the unfolding of loops.
- `Cyclic unfolding` :
  It is an integer, which enables to get more precision on the fixpoint computation in loops by virtually

unrolling them : it allows us to compute the fixpoint of a more contracting operator, and obtain a less perturbed iteration, by doing the join operation less often. It is complementary to the initial unfolding.

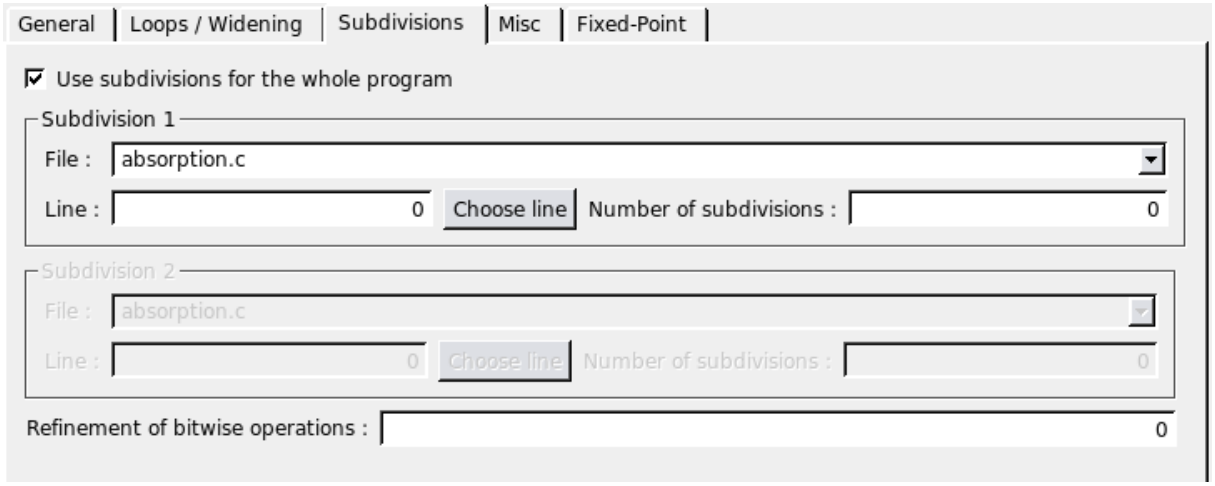
- `Keep relations at the end of loops` :
  When the box is ticked, all relations computed in the loops are kept, otherwise they are agglomerated at the exit of the loop (the idea being that some relations needed in the loop for good results are no longer necessary after the loop). In the case when these relations are not kept, the computation will generally be less costly but less precise.
- `Global union` :
  Experimental feature to improve union operator. This option should not be used for the time being.
- `Unfold non-relational errors` :
  When the box is ticked, more information is computed about the iteration in loops where the errors occur - more costly with same precision, but more information may be displayed at the user interface.
- `Automatic Parameters` :
  In this mode, the analyser tries to choose "optimal" values for the number of executions before unions and number of unfolding of the loops, for each loop computation, that should allow to get tight ranges for the errors. When this parameter is set, the user can choose the `Maximum Iterations` parameter. Also some the parameters in the `Loops` sub-menu (except for `initial unfolding`) can still be set, they are those used for the computation of the values of variables.
- `Maximum Iterations` :
  Integer parameter, setting the maximum number of iterations before deciding a choice of unfolding in the automatic mode. This option is available only when the `Automatic Parameters` mode is set.
- `Widening Threshold` :
  It is an integer that indicates after how many iterations the analyser will begin to use a widening operator. The larger this parameter is, the more precise (and time consuming) the analysis gets. When a loop is unrolled, the `Widening Threshold` parameter indicates how many times the new functional is executed before using widening operators. It is thus equivalent to using widening operators after `Widening Threshold * Unfold Loop Number` iterations of the real loop.
- `Narrowing Threshold` :
  No narrowing is implemented for the time being.

- `Progressive reduction of precision` :
  Check this parameter box to enable a progressive reduction of precision in computations occuring in loops. The idea is that when loop computations should converge at infinite horizon in real numbers, reducing the precision of computations (by decreasing the mantissa size in the analysis) will accelerate the convergence. Then, the three following parameters are taken into account.
- `Number of iterations before reduction` :
  We first do some number of iterations with full precision.
- `Loss of precision (bits)` : n every x iterations
  Caracterizes the loss of precision in the mantissa size : n bits will be lost every x iterations.
- `Minimal precision` :
  Sets the minimum number of bits of precision (when it is reached, we iterate the computation with same precision).
- `Exact error calculation for constants` :
  When the box is ticked, the rounding errors are always computed as accurately as possible : when the floating-point result of an operation is a floating-point number (and not an interval), the error can thus be bounded taking as reference the same computation with higher precision floating-point numbers (MPFR library). Otherwise (if the box is not ticked), this is still the case outside loops, but for the computation or propagation of errors in loops, we use the `ulp` error, which is less accurate but often allows a better convergence of fixpoint algorithms.
- `Convergence of higher order errors` :
  When the box is ticked, the convergence in fixpoint computations is decided on each first-order error

and the higher order error. But it has been noted that higher order errors often seriously delay the convergence of fixpoint computation, while being completely negligible. For this reason, we propose a convergence criterion based on each first-order error and the global error. The convergence is better, but higher order errors are over-approximated. The semantics of operations would have to be modified to suppress this inconvenience.

Subdivisions :

The options proposed here for the time being are used to improve the accuracy of results, by subdividing intervals to reduce the problem of lack of correlations in the analysis. Three different and independent ways of using subdivisions are proposed. The global subdvisions or subdivisions for the whole program represents the main way. They are available from the user interface. By this way, the user divides one or two global intervals into regular slices. These subdivisions may be useful with both the relational and the non-relational analyses. The manual or local subdivisions are introduced in this section and described in details in section 3.9. Bitwise subdivisions is the last way to do subdivision, especially for integer variables (see section 4.3.3).



– `Use subdivisions for the whole program` :
Using large intervals as inputs can lead to large over approximations of the results, even with the relational abstract domains (particularly when non affine operations are used, divisions, high degree polynomials, etc). Thus, if the user still wants to analyse a program for a large set of inputs, we give the possibility to subdivide the input interval, execute as many partial analyses as the number of subdivisions indicates, and present as final result the union of all these analyses.
When this option is selected, we give the user the possibility to divide at most two intervals. Indeed, the number of analyses is then the number of subdivision of first interval multiplied by the number of subdivisions of the second interval, in order to take all possibilities into account.
The intervals to be subdivided can be selected either by giving the source file and the number of the line in the source code (beware, the first line of the source code is noted as line number 2). Or, more simply by clicking on the `Choose line` button, and selecting the line in the program by clicking.

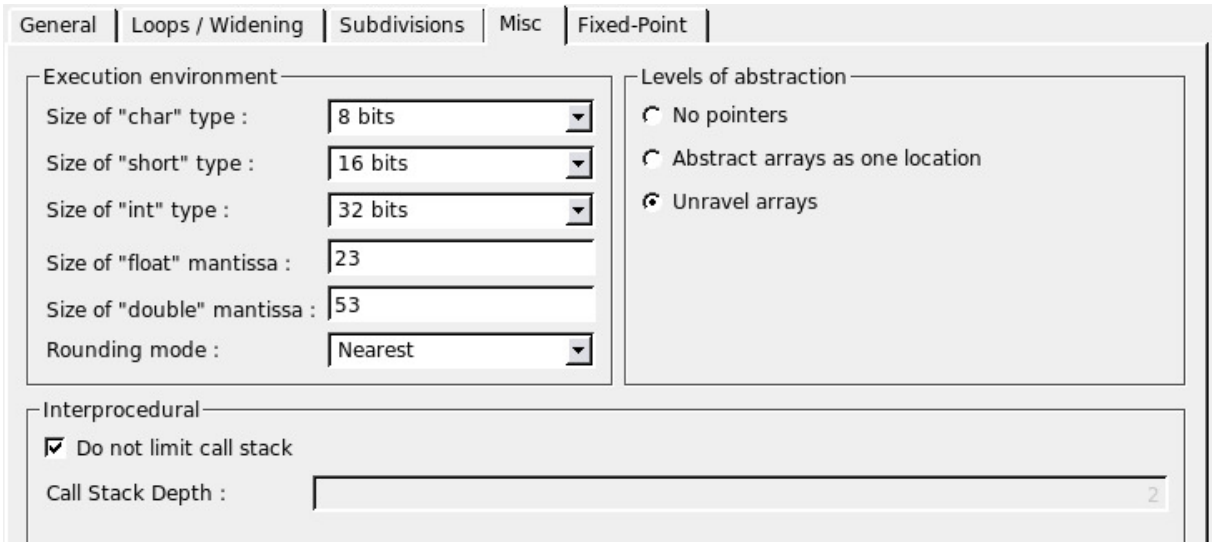
– `Local subdivisions` :
Local subdivisions have been introduced for a finer control of the subdivision mechanism. They enable to subdivide a part of the code, like a `switch/case` construction whose `case` are sub-intervals for the subdivided variable.
Fluctuat proposes four kinds of local subdivisions :
  – regular subdivisions with the keywords `SUBDIV_BEGIN` and `SUBDIV_END`,
  – irregular subdivisions with the keywords `SUBDIV_STEP_BEGIN` and `SUBDIV_END`,

- subdivisions to meet a requirement with the keywords `SUBDIV_REQUIRE_BEGIN` and `SUBDIV_END`,
- manual subdivisions within loops with the keywords `DBETWEEN`, `DCOLLECT`, `DCOLLECT_WORST`, `DJOIN`

– `Refinement of bitwise operations` :
We have noted that the propagation of existing errors through bitwise operations could lead to huge over approximations when the interval values for the inputs have an important width. We thus give the possibility to subdivide these values for the computation of errors (by an integer number of subdivisions), the results are more accurate, but the cost in memory and computation time can be heavy. However, the subdivision is only local to bitwise operations, and only one analysis is performed. An example of the behavior of this parameter is given in Section 4.3.3

Misc :

If the program you want to analyse will not be used on a `IEEE754` compliant processor, you can change the behavior of Fluctuat to specify your processor types. The section `Execution environment` allows you to select the specifications of your processor.
Fluctuat contains three methods to represent alias (pointers and arrays). You can choose one of these in the section `Levels of abstraction`.
To analyze recursive functions, you need to limit the depth of the function calls interpreted by Fluctuat. You can control this behavior in the section `Interprocedural`.



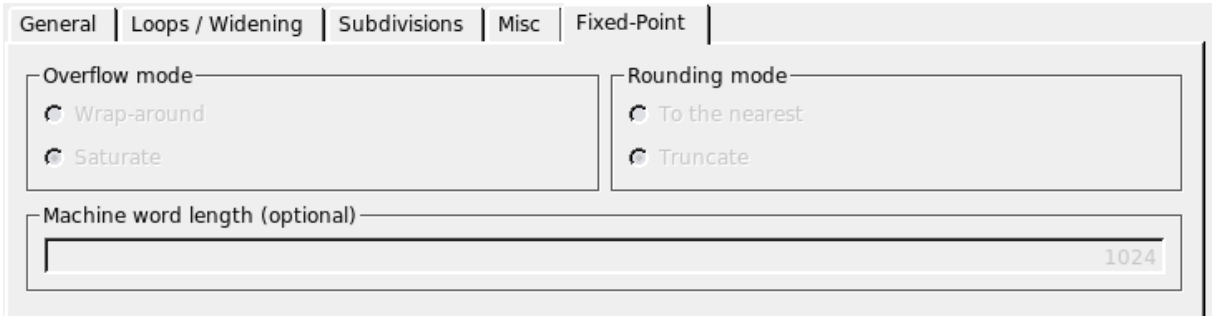
– `Size of ‘‘char’’, ‘‘short’’ and ‘‘int’’ type` :
The choice of the number of bits used for the representation of each integer type (signed or unsigned `char`, `short` or `int`), is left to the user, except for `long int`, which are necessarily represented on 32 bits. Also, the `int` types must be represented on at least as many bits as `short` types, which must themselves be represented on at least as many bits as `char` types.
– `Size of ‘‘float’’ and ‘‘double’’ mantissa` :
These parameters represent respectively the number of bits of the mantissa in the format in which float and double numbers are represented. For example for a double in IEEE754 norm, the size of double mantissa is 53.
– `Rounding mode` :
This parameter indicates how your processor rounds results of its computations. You can select one of the four possible modes defined in the norms.
– `No pointers` :

Aliases (arrays, pointers) are not interpreted. This option must be preferred to the others when the programs being analysed contains no alias, because the analysis is quicker and uses less memory. One must be careful : if ticking this box when a programs does contain aliases, the analysis may produce no error, while giving false results.

- `Abstract arrays as one location` :
  Aliases are interpreted, but only one value agglomerates all coefficients of an array. However, the fields of structures are not agglomerated. The analysis may be imprecise in presence of arrays.
- `Unravel arrays` :
  Aliases are interpreted, and one value is associated to each coefficient of an array. The analysis may be costly.
- `Do not limit call stack` :
  When the box is not ticked, the size of the call stack is limited by the integer `Call Stack Depth`. Otherwise, it is not limited, but only the last call is kept in memory (indeed, the others are not needed if we are interested mainly by the result at the end of the program). The results are most of the time more accurate and got faster, but limiting the call stack is still useful in some cases, for analysing recursive functions for example.
- `Call Stack Depth` :
  An integer which should not be too big. It describes the size of the abstraction of the call stack (see Section 4.3.2). As a consequence, the bigger the call stack depth, the more precise the analysis gets, since functions are then analysed separately for different contexts. The analysis gets also more costly in terms of time and memory consumption. Of course, this parameter can be set only if the box `Do not limit call stack` is not ticked.

Fixed-Point :

This section details the parameters of the fixed-point mode :

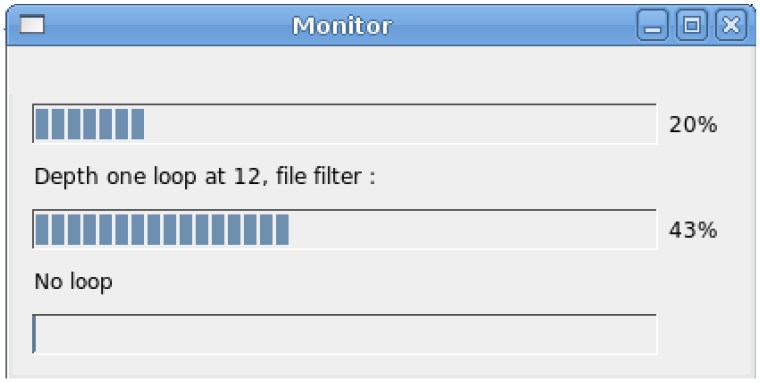


- `Overflow mode` :
  This option changes the behavior of the analyzer when an operation overflow the format. The first mode `Wrap-around` indicates the analyzer to wrap around the maximum - or minimum - value of an integer (set with the option `Machine word length`). The second mode `Saturate` indicates to set the value to the maximum - or minimum - value of an integer.
- `Rounding mode` :
  This option changes the rounding method. The default value `Truncate` rounds all operations to zero. The value `To the nearest` rounds all operations to the nearest (caution 2.5 is rounded to 3, not to 2).
- `Machine word length` :
  This integer represents the maximal number of bits the processor can use in integer computations. The common sizes are 32 or 64. This value must be greater or equal to size of float mantissa.

2.4.4 Monitor

During the analysis, you can view the progress by two means :
– The small progress bar in the right bottom of the main window,
– The full progress window accessible from the action `View monitor`.

The small bar is a general progress view who mix the progress of subdivisions (if present) and progress of loop iterations.



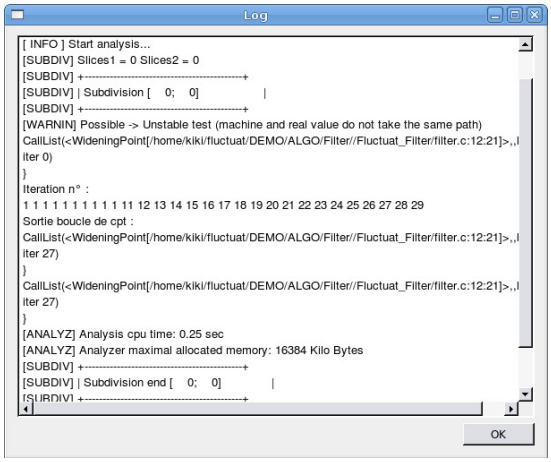
In the full window, you can view the progress of subdivisions in the first line, the progress of first loop iteration in the second line, and the progress of second loop in the third line. Each line is enabled only if subdivisions or loops are presents. The progress of loop iterations is scaled of the maximum possible iterations. Then the analysis would reach 100% only in the worst case.

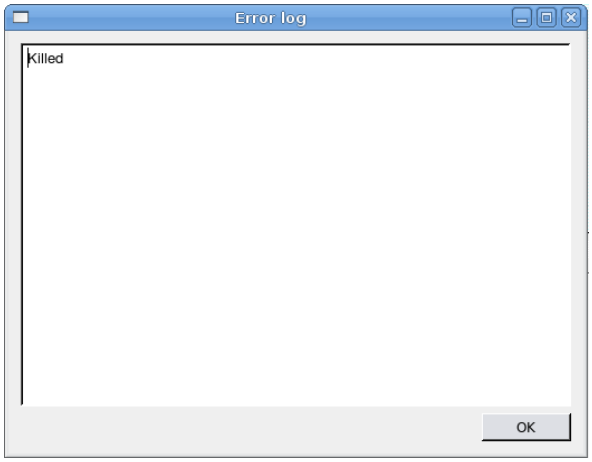
2.5 Visualisation of the results

Once the analysis done, you can inspect the all the results with the graphical interface. The following sections explain how to find any useful information about the analysis.

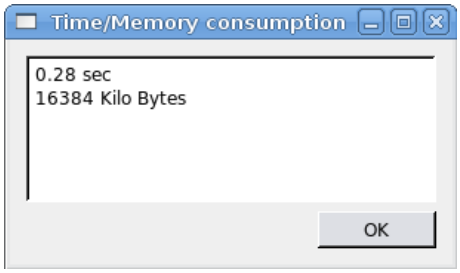
2.5.1 Analysis information

With the actions `View log` and `View error log`, you can see the normal and error outputs generated by Fluctuat during the analysis :



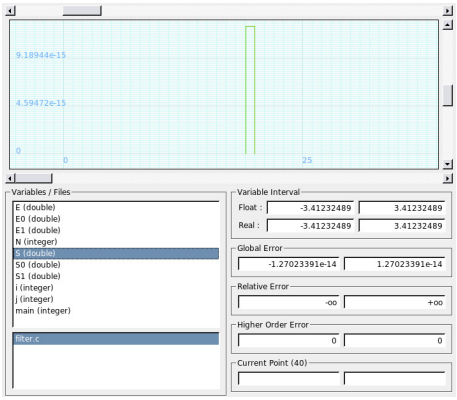


Some statistics about the analysis can be viewed in the right bottom of main window, and with the `View stat` action.



2.5.2 Variable values and errors

The main window of the analyser displays the list of identifiers [2] occurring in the abstract environment at the end of the analysis and a graph representation of the abstract value related to the selected identifier in the list. The floating point value, the real value, the error and the relative error of the selected identifier are displayed in the right of the list.



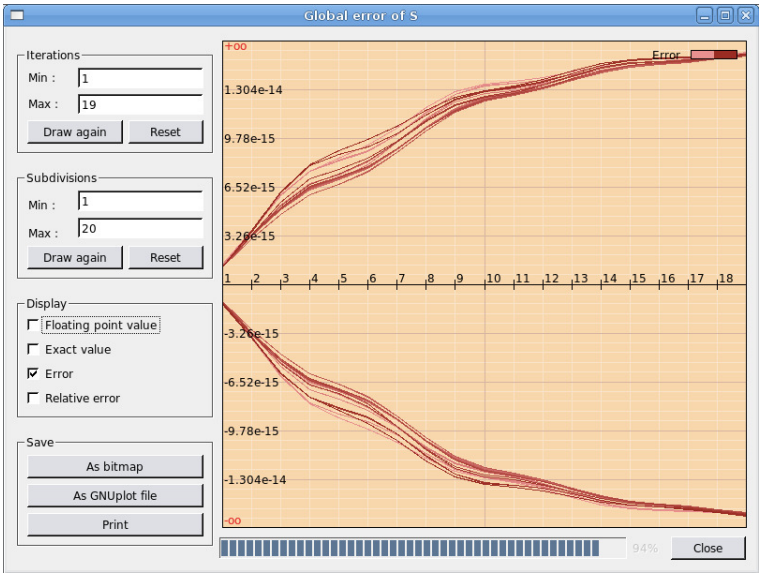
View information about a variable can be achieved by selecting the variable in the list, or by clicking on a link in the source code window. Once a variable is selected, the graph represents the error series of a variable *id* and thus shows the contribution of the operations to the global error on *id*. The operations are identified to their program point which is displayed on the X-axis. Scroll-bars on the sides of the graph window are used to do various kinds of zooms. A `Reset Graph` button allows to come back to the default places of the scroll-bars.

The bars indicate the maximum of the absolute values of the interval bounds. This enables to assert the correctness of the code, when the errors have a small magnitude. The graph and program code frames are connected in the graphical user interface in such a way that clicking on a column of the graph makes the code frame emphasises the related program line. This enables the user to easily identify which piece of code mainly contributes to the error on the selected variable. Also, when a given variable is selected, the user can know which lines mainly contribute to its error, using the `Biggest error` and `Next error` buttons on the top of the main window. When choosing the biggest error, the corresponding line in the C program window is highlighted. Then, each time the user clicks the `Next error` button, the line corresponding to the next error in terms of maximum magnitude is highlighted.

2. The list of identifiers is automatically sorted in alphabetic order.

2.5.3 Evolution graphs

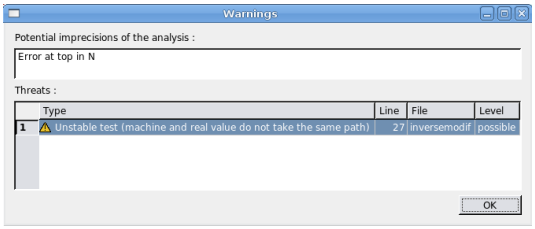
From the interface, one can draw, for a variable selected in the variable list, the evolution over iterations of the value, the global error, or the global relative error. For this to be possible for a double variable `S` for instance, the user has to use the assertion `DPRINT(S)` in the program, so that the values and errors of variable `S` are logged each time the analyser meets this assertion (see Section 3.7). This will open a new window containing the relevant graphics, as shown below.



Once the window open, the chosen graph is displayed. In the left of the window, you can choose the iterations and the subdivisions you want to display. You can also superimpose different view of the variables with the checkboxes below. All these graphs can be printed , or saved as a bitmap or gnuplot file.

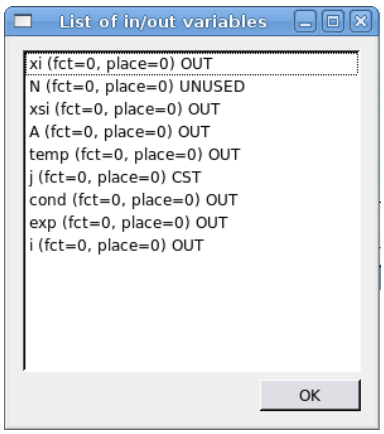
2.5.4 Warnings

The warnings about potential run time errors, such as division by zero or the square root of a negative number, are recalled in a separate window, by selecting `View warnings`.



Clicking on one error of this window selects, in the source file, the line at which this potential error occurred. The idea is not to detect run time errors, as we suppose there are none and go on with the analysis, but to point out an imprecision of the analysis which should if possible be settled, for example using assertions, in order to get more accurate results. In the same way, the warnings about possibly unstable tests and infinite variables are also displayed. A full list of examples is given in Section 4.5.

2.5.5 In/out variables

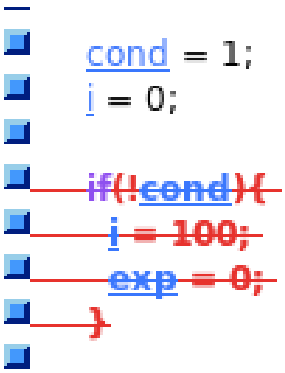


If the `In/out analysis` mode was selected when setting parameters of the analysis, one can also check the in/out status of variables. This is useful in particular to analyse parts of big programs, to check that there are no uninitialised inputs, and to spot outputs.

When selecting `View in/out` in the `Analysis` menu, a window opens, in which all variables are printed with one of the following values :

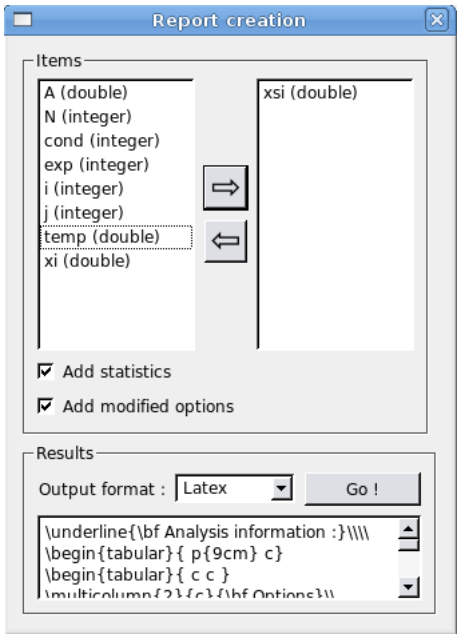
- `CST` : constant
- `initialised IN (Builtin)` : variable which first use was an initialisation using a Builtin
- `UNUSED` : unused variable
- `non initialised but used IN` : variable used without being first initialised : a value or an assertion should be added.
- `OUT` : any other case

2.5.6 Dead code



This feature is experimental and must be used very carefully. By selecting the `View dead code` action, the code blocks unvisited by the analyzer are crossed out in red in the source code window.

2.5.7 Save results



A useful way to create report is to save chosen information about the analysis parameters and the results. It can be done with the `Dump results` action.
You have to select variables you want to save. You can also check boxes if you want to add analysis parameters and analysis statistics. And finally, you choose the output format [3] and click on the `Go` button.
The text you want appears in the below window and you just have to copy/paste it in your report.

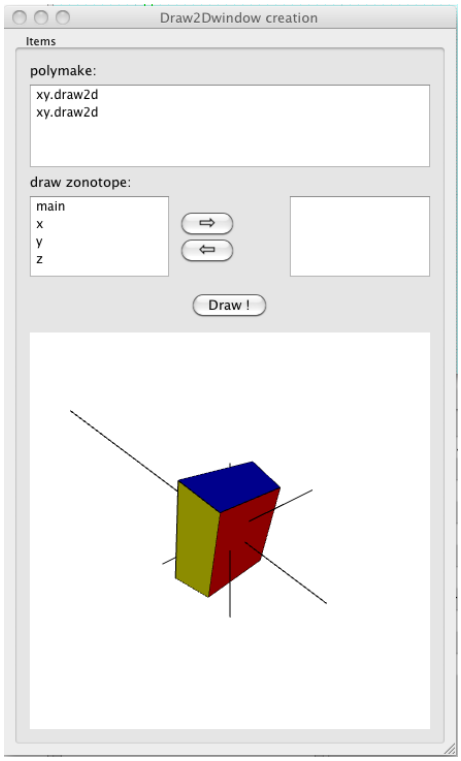
2.5.8 Draw results (2D/3D)

You can visualize the zonotopes representing the value of 2 or 3 progam variables with the `Draw results` action. This action is possible when you previously defined some builtins in C code, namely the builtin `DRAW2D,DRAW3D,DAFFPRINT`. The parameters of these builtins are the names of respectively 2, 3 and 1 program variables.

`DRAW2D` and `DRAW3D` generate an output file that contains Polymake[1] commands. You then have to use Polymake to visualize the zonotopes.

Fluctuat also provides its own method to visualize zonotopes. You first have to record the affine form of some program variables (`DAFFPRINT` builtin). This is done automatically during the analysis. In the 'Draw Result (2D/3D)' window, you then have to select the variables (2 or 3) you want to visualize, then push on the `Draw` button.

3. For the time being, there is only two outputs mode `latex` and `html`



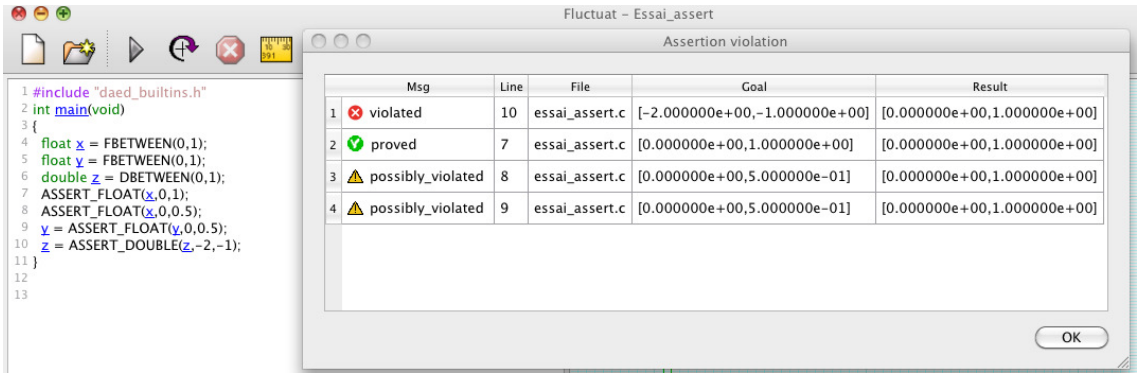
The following lines of code produce the example shown on the figure above.

```
double x = FBETWEEN(0,1);
double y = FBETWEEN(0,1);
double z = DBETWEEN(0,1);

z=x*x-y*y+1;
DAFFPRINT(x);
DAFFPRINT(y);
DAFFPRINT(z);
```

The zonotope is drawn in the below window and you can rotate it with the mouse (hold left-click and move).

2.5.9 Show Assertions

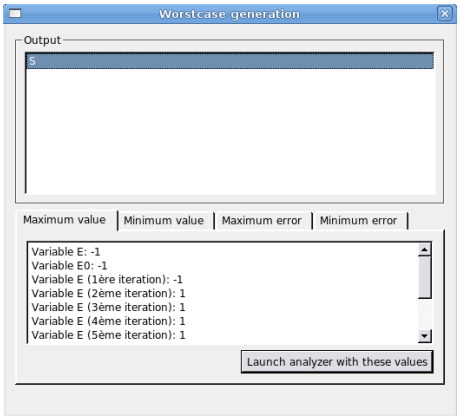
You can define assertions on the value of program variables. An assertion is the name of a variable and two bounds of an interval. An assertion $x \in [a, b]$ is proved if Fluctuat determines that the value of x is always between a and b. The assertion is violated if Fluctuat determines that the value of x is always between below a or above b. Otherwise, its is possibly violated. Depending on the type of x, you need to use the builtins `ASSERT_FLOAT` or `ASSERT_DOUBLE` to define an assertion, as shown on picture blow :


Fluctuat – Essai_assert
Assertion violation
1 #include "daed_builtins.h"
2 int main(void)
3 {
4 float x = FBETWEEN(0,1);
5 float y = FBETWEEN(0,1);
6 double z = DBETWEEN(0,1);
7 ASSERT_FLOAT(x,0,1);
8 ASSERT_FLOAT(x,0,0.5);
9 y = ASSERT_FLOAT(y,0,0.5);
10 z = ASSERT_DOUBLE(z,-2,-1);
11 }
Msg | Line | File | Goal | Result
1 violated | 10 | essai_assert.c | [-2.000000e+00,-1.000000e+00] | [0.000000e+00,1.000000e+00]
2 proved | 7 | essai_assert.c | [0.000000e+00,1.000000e+00] | [0.000000e+00,1.000000e+00]
3 possibly_violated | 8 | essai_assert.c | [0.000000e+00,5.000000e-01] | [0.000000e+00,1.000000e+00]
4 possibly_violated | 9 | essai_assert.c | [0.000000e+00,5.000000e-01] | [0.000000e+00,1.000000e+00]
OK

If some assertions are defined, this window will pop up automatically when the analysis terminates. For more details, see Section 3.4.

2.5.10 Worst-case generation

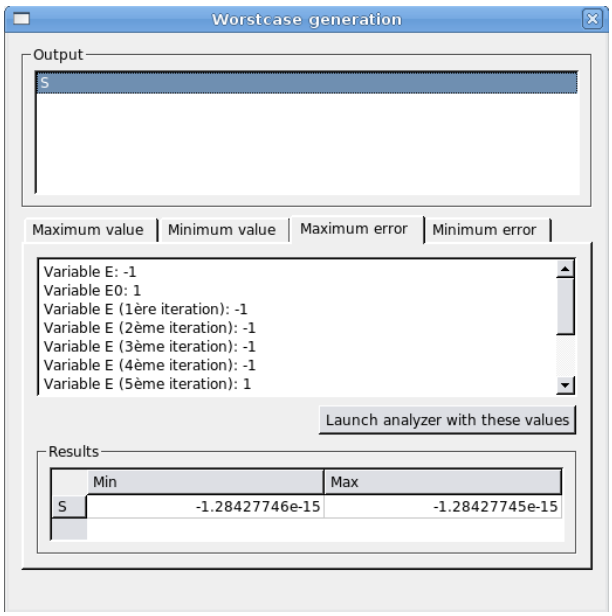
The two following options only work with the `Fully relational (global error)` analysis mode.



Sometimes, it is useful to generate input values who lead to worst output values and errors. You can obtain such input values with this feature.
First you need to insert a `DSENSITIVITY(x)` assertion (where `x` is the name of the output value you want to generate). Analyze your project, and after the analysis, the `Generate worst reachable case` window should display this variable. By selecting it, Fluctuat will produce lists of values for the interval input variables (only the `DBETWEEN` variables) :

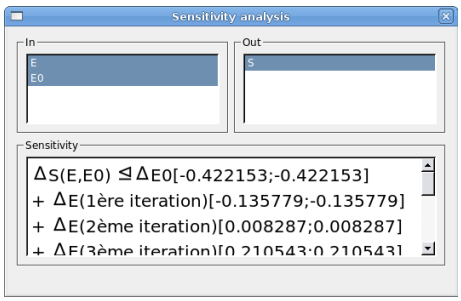
- the first list (in the `Maximum value` section) will lead to the maximum output value,
- the second list to the minimum value,
- the third list to the maximum error,
- and the fourth list to the minimum error.

You can try these values with buttons in the bottom of each list. It will launch an analysis with the worst-case input values instead of the input intervals.
The results will be shown at the end of analysis above the button :



When you use manual subdivisions in your project, you can also generate worst-cases with the builtin `DCOLLECT_WORST`. Its use and an example is given is Section 3.8.
For best results, it's important to keep the parameter "Internal accuracy" (or "MPFR bits" in advanced mode) in a high value.

2.5.11 Sensitivity analysis



With the same assertion (`DSENSITIVITY`), you can obtain information about the decomposition of the output values. As shown in the below picture, if you select an output variable, and some input variables. You can see the propagation in the output of a input variation :

$$\Delta x(e1, e2) \trianglelefteq \Delta e1[a; b] + ...$$

In this example, if the input `e1` variates with a value `i` the output `x` variates with a value $i * [a; b]$.

2.6 Batch mode

Fluctuat can be used in batch mode. For this purpose, you have to set the environment variable `FLUCTUATHOME` to the directory containing your Fluctuat distribution (1), and add Fluctuat `bin` directory to your `PATH` (2) :

```
(1) bash> export FLUCTUATHOME=_path_to_fluctuat_directory_
(2) bash> export PATH=$FLUCTUATHOME/bin:$PATH
```

2.6.1 Command-line scripts

Two scripts can be used :
– `daed_project`, to build a Fluctuat project and pre-process your source files,
– `daed_analyze`, to launch an analysis.

Script "daed_project" :
The usage of this command is :

```
daed_project [options] file1.c [file2.c ...]
```

The "options" are `gcc` pre-processing options like `-D` or `-I`. For each source file containing Fluctuat assertions, the following directive must be included :

```
#include "daed_builtins.h"
```

This command will constructs :
– a `project.xml` file containing information about the project,
– a `graph.dot` file with the graphical representation of the program in `GraphViz`[4] format.
– `logfile` and `console.eout` files with the normal and error outputs of the internal commands.
– and a `.ii` pre-processed file for each `.c` file.

Information about the program and included files are saved in the `project.xml` file (see syntax in Section 2.6.2).

Script "daed_analyze" :
The usage of this command is :

```
daed_analyze [options] file1.ii [fil2.ii ...]
```

The `.ii` files are the pre-processed files produced by the previous script. The options can be :
– `-v` for verbose mode,
– `-o _dir_` to change the default output directory (default is "output"),
– `-c _resource_file_` to change the default resource file with (default is "project.rc").

The analyse parameters are provided by the resource file, according to the grammar specified in the appendix E. This command will creates a directory who contains information about program and resulting variables. In this directory, the file `liste_variables.xml` contains the list of all variables saved at the end of the analysis. For each variable, a file `_variable_name_.var` contains intervals for floating-point, real and error values. File `time_memory.out` contains time and memory amounts used by the analysis. The first line of each dead code block is saved in the file `code_mort.out`. Warnings and alarms are saved in files `warnings.xml` and `liste_infinis.xml`.

If assertions `DPRINT(_variable_)` are used, the saved values are in the files `_variable_.evolval` for the floating-point values, `_variable_.evolerr` for the errors and `_variable_.evolvalexact` for the real values. In these files, each line represents the interval when Fluctuat interprets the assertion.

2.6.2 Input and output files

Syntax for "project.xml" :
The Relax NG compact syntax of this file is :

4. Graph Visualization Software - http ://www.graphviz.org/

```
start = project

project = element project { attribute name {string},
                            attribute language {string},
                            originalFiles*,
                            workingFiles*,
                            includedFiles*,
                            entryPoints* }

originalFiles = element originalFiles { file* }
workingFiles = element workingFiles { file* }
includedFiles = element includedFiles { file* }
entryPoints = element entryPoints { function* }

file = element file {attribute name {string} ?,
                     attribute path {string}}
function = element function {attribute name {string}}
```

An example of this file :

```
<project name="Inversemodif" language="C">
        <originalFiles>
                <file name="inversemodif"
                      path="/Users/ktekkal/fluctuat/DEMO/ALGO/Inverse/inversemodif.c"/>
        </originalFiles>
        <workingFiles>
                <file name="inversemodif.c" path="inversemodif.c"/>
        </workingFiles>
        <includedFiles>
                <file path="/Users/ktekkal/fluctuat/ABSINT/two_include/daed_builtins.h"/>
        </includedFiles>
        <entryPoints>
                <function name="main"/>
        </entryPoints>
</project>
```

Syntax for "liste_variables.xml" :

```
start = variables
variables = element variables { variable * }
variable = element variable { name, type }
name = attribute name { string }
type = attribute type { "integer" | "float" | "double" }
```

An example of this file :

```
<variables>
        <variable name="temp" type="double"/>
```

```
        <variable name="xi" type="double"/>
        <variable name="xsi" type="double"/>
        <variable name="A" type="double"/>
        <variable name="cond" type="integer"/>
        <variable name="i" type="integer"/>
        <variable name="exp" type="integer"/>
        <variable name="j" type="integer"/>
</variables>
```

Syntax for ".var" files :
For each variable, a file `_variable_name_.var` contains :

- In the 1^{st} line, the floating-point interval,
- In the 2^{nd} line, the real interval,
- In the 3^{rd} line, the global error interval,
- In the 4^{th} line, the relative error interval,
- In the 5^{th} line, the higher-order error interval,
- In the following lines, the decomposition of errors until the token "enderr", each line represents a contribution to the global error, with information about the position of this contribution in the program (line, file, function and interval),
- In the next lines, the evolution of the errors along the iterations (line, iteration, epsilon, file, function and interval).

And for example with test "DEMO/SIMPLE/Assoc" :

```
0 6.00000000                                                  # Floating-point interval
0 6.00000000                                                  # Real
-4.44089210e-16 4.44089210e-16                                # Global error
-oo +oo                                                       # Relative error
0 0                                                           # Higher order error
1 associativite.ii main -4.44089210e-16 4.44089210e-16     # Error decomposition
enderr
1 0 0 associativite.ii main -4.44089210e-16 4.44089210e-16 # Iteration decomposition
```

Syntax for "warnings.xml" :

```
start = warnings
warnings = element warnings { warning* }
warning = element warning { type, level, message, file, line }
type = attribute type { "division" |               # Division by zero
                        "log" |                    # Logarithm of a negative value
                        "sqrt" |                   # Square root of a negative value
                        "conversion" |             # Undefined float-to-int conversion
                        "unstable" |               # Unstable test
                        "unstable_real" |          # Unstable test (real is bottom)
                        "unstable_float" |         # Unstable test (float is bottom)
                        "builtin_bounds" |         # Builtin bounds not exactly represented
                        "program_never_end" |      # Program that never end
```

```
                        "rounding_error_missing" | # Rounding error to specified in norm
                        "dcollect_hole_real" |     # Warning for assertion DCOLLECT_WARN
                        "dcollect_hole_float" |    #  ''
                        "alias_error" |            # Bad pointer location
                        "other" }                  # See message for this warning
level = attribute level { "possible" | "sure" }
message = attribute message { string }
file = attribute file { string }
line = attribute line { string }
```

and for example :

```
<warnings>
  <warning type="unstable" level="possible"
     message="Unstable test (machine and real value do not take the same path)"
     file="/Users/ktekkal/fluctuat/DEMO/ALGO/Inverse/Fluctuat_Inversemodif/inversemodif.c"
     line="22"/>
</warnings>
```

Syntax for "liste_infinis.xml" :

```
start = infinity
infinity = element infinity { variable * }
variable = element variable { name, type }
name = attribute name { string }
type = attribute type { "value" | "error" | "higher-order error" }
```

and for example :

```
<infinity>
        <variable name="temp" type="value"/>
        <variable name="temp" type="error"/>
        <variable name="xi" type="value"/>
        <variable name="xi" type="error"/>
        <variable name="xsi" type="value"/>
        <variable name="xsi" type="error"/>
        <variable name="i" type="value"/>
</infinity>
```

2.7 Interactive static analysis

Fluctuat can perfom static analyses in an interactive mode. The objective of this mode is to explore the parameters and the annotations in order to make the automatic mode succeed in filling the numerical requirements.

The interactive mode presents a user's interface close to the gdb's one. Hence most of the command are inherited from gdb. The differences from gdb are the following :

– the interactive static analysis can start from any entry point ;
– the analysis can explore several branches/paths in the same session ;
– the result of print command are intervals with accuracy errors instead of simple values.

The interactive mode is invoked in a separate process and it communicates through a pipe with the graphical user interface. The user can kill the process by typing “Ctrl+K” in the user interace or by killing the “interactive_analysis” processes from the command line. All the commands of the interactive mode are executed in the “Fluctuat_...” project directory.

2.7.1 Invoking the interactive mode

To make the interactive mode available, the user has first to create or to load a project (see the section /refcreateProject and the section 2.3.2). To invoke the interactive mode, the user can type “Ctrl+D” or click on the button “start interactive analysis” in the “Analysis” menu bar. Then a new text window appears (see figure 2). It is in direct connection with an active analysis.

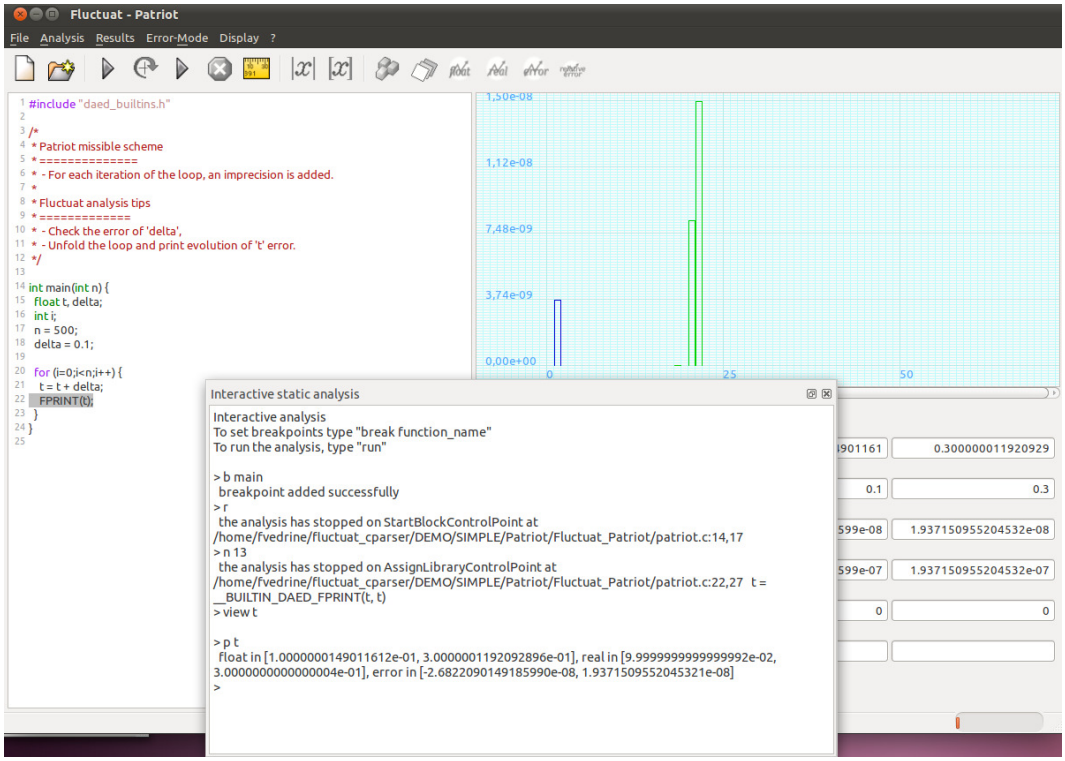


FIGURE 2 – Interactive Static Analysis Window

At this point, an interactive analysis contains three steps : the configuration step, the analysis step, the result exploration step. The commands of each step are detailed in the next section. Typing an empty command replays the previous command. The commands are executed when the user enters the “enter” key. Then the executed command is the line where the “enter” key occurs.

The configuration step consists in setting the permanent breakpoints – with the `break` command –, in setting the optional resource file – with the `setr` command – and in setting the entry point of the analysis – with the `run` command. This last command changes the current step to the analysis step.

The analysis step consists in covering the function called from the entry point. Each time the analysis encounters a breakpoint it stops and propose the user to enter commands. The available command are `step`, `next`, `finish`, `continue` to abstractly interpret one instruction, one instruction or a call to a function, several instructions up to end the current function, several instructions up to the next breakpoint. The `print` command enable to print the content of a local or global variable as an abstract domain. When the analysis reachs the end of the function called from the entry point, it enters the result exploration step.

For the moment the result exploration step automatically displays the content of each local variable of the function called from the entry point and the content of each global variable at the end of the interactive analysis.

2.7.2 commands of the interactive mode

`break function_name` **or** `b function_name :` add a permanent breakpoint on the entry point of `function_name`.

`break file_name:line` **or** `b function_name :` add a permanent breakpoint on the line `line` in the file `file_name`.

`setr resource_file :` load the analysis parameters from the file `resource_file`. As the analysis executes in the "Fluctuat_..." project directory, `setr project.rc` loads the analysis parameters of the last static analysis launched.

run or `r :` launch an interactive static analysis from the "main" function. If such a function does not exist, the command fails.

`run function_name` **or** `r function_name :` launch an interactive static analysis from the "function_name" function. If such a function does not exist, the command fails.

`stepi :` "Step" one atomic instruction of the fluctuat analyzer. For example, if an instruction `int x = f() + g();` appears in the code, then $n + m + 3$ `stepi` commands are required to store the abstract value in the variable `x`, where n is the number of atomic instructions in f and m is the number of atomic instructions in g. This point comes from the fact that in Fluctuat such an instruction is decomposed into `int temp_0 = f();`, followed by `int temp_1 = g();`, followed by `int x = temp_0 + temp_1;`.

`stepi count :` "Step" `count` atomic instructions of the fluctuat analyzer, following the `stepi` exploration policy. This command is especially interesting to go backward a special event. Combined with the `info count` property that displays the number of atomic instructions from the beginning of the analysis, it is simple to rerun the analysis with a smaller number of instructions. For example, the commands

```
> break main
> break interesting_function
> run
> continue
> info count
  xxx instructions have been executed
Ctrl+K
Ctrl+D
> break main
> run
> stepi xxx-5
```

will stop before the `interesting_function`.

`step` **or** `s :` "Step" one instruction of the fluctuat analyzer. If the instruction under analysis is a call instruction, then the analysis enters the call. For example, if an instruction `int x = f() + g();` appears

in the code, then $n+m+1$ `step` commands are required to store the abstract value in the variable `x`, where n is the number of instructions in f and m is the number of instructions in g.

`step count of s count :` "Step" `count` instructions of the fluctuat analyzer, following the `step` exploration policy.

`next or n :` "Next" one instruction of the fluctuat analyzer. If the instruction under analysis is a call instruction, then the analysis covers the whole function called to reach the next line after the current instruction. For example, if an instruction `int x = f() + g();` appears in the code, then only one `next` command is required to store the abstract value in the variable `x`.

`next count of n count :` "Next" `count` instructions of the fluctuat analyzer, following the `next` exploration policy.

`finish or fin :` "continue" to interpret the current function until exiting.

`finish count or fin count :` "continue" the abstract interpretation of the code, up to have popped `count` functions from the current stack. In practice, this command applies `count` times the `finish` command.

`continue or c :` "continue" to abstractly interpret the code until the next breakpoint.

`info count :` return the number of atomic instructions covered since the beginning of the analysis.

`break or b :` add a permanent breakpoint on the current line in the interactive session. The user should have launch the analysis with the `run` command.

`break if t == 5 or b if t == 5 :` add a conditional permanent breakpoint on the current line in the interactive session. The variable `t` should be active at the current line and it should have a numeric type. The analysis will next stop on the current control point if the domain of `t` contains the value 5.

`break if t is in [3,5] :` add a conditional permanent breakpoint on the current line in the interactive session. The variable `t` should be active at the current line and it should have a numeric type. The analysis will next stop on the current control point if the domain of `t` has a non-empty intersection with the interval $[3, 5]$.

`break if t is top :` add a conditional permanent breakpoint on the current line in the interactive session. The variable `t` should be active at the current line and it should have a numeric type. The analysis will next stop on the current control point if the domain of `t` is widened to *top*.

`delete or del :` remove all the permanent breakpoints.

`print variable_name or p variable_name :` display the abstract value stored in the variable `variable_name`. This diplay provides the implementation range of the variable, the real range of the variable and the error attached to the variable. As an example,

```
> print a
> print b
> stepi
> print c
```

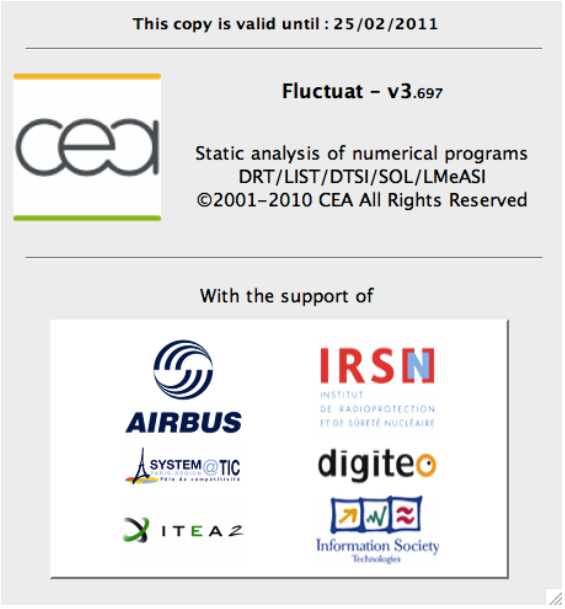
means that before interpreting an instruction `c = a+b;` the abstract values of `a` and `b` were in the described ranges and after interpreting the instruction the absract value of `c` is in the described range for `c`. Since the interactive analysis does not display the internal format with all the relations, it is false to interpret the displays as if the abstract values of `a` and `b` were in the described ranges then the value of `c` is in the described range for `c`.

`view variable_name :` display the internal format – with dependencies resolved with respect to the lines of the source code – of the abstract value stored in the variable `variable_name`. The result is visible on the graphical fluctuat window.

`draw var_name_1, var_name_2(, var_name_3) :` graphically display some relational information between the variables `var_name_1`, `var_name_2` and `var_name_3` if this last variable is available. The displayed geometrical form is a zonohedra in dimension 3 or 2. The direction for which the thickness of the zonohedra is the least corresponds to interesting relational information that can give a precise invariant.

2.8 License

The use of Fluctuat is limited in time by a license file called `fluctuat.lic`. This file must be saved in the Fluctuat home directory specified by the `FLUCTUATHOME` environment variable. You can see the expiry date of your Fluctuat copy in the `About Fluctuat` window.
This window contains also information about the partners who support this project.



3 Assertions in the source code

A number of assertions can be added to the analysed source code, to help the analyser through interactions with users, to give some hypotheses on the programs, or to print information on evolution of variables in the program.
We first detail the assertions that allow to give information on variables, then those used to print information on evolution of variables in the program (that can than be graphically visualized - see Section 2.5.3). These assertions are simply inserted by hand in the source code. All assertions have names that specify the type of the variable considered (names including, depending on the type of assertions, `D` for double precision, `F` for floating-point precision and `I` for integers, or the complete keywords `DOUBLE`, `FLOAT`, and `INT`).

We are aware that the length of current names of the assertions is quite tedious. That is why we defined aliases to all following assertions that allow to omit the `__BUILTIN_DAED_` part. For example, the two following assertions are equivalents :

```
x = DBETWEEN(0,100);
```

and

```
x = __BUILTIN_DAED_DBETWEEN(0,100);
```

3.1 Specifying ranges on values of variables

When you specify a variable only by a range of value, we consider there is no representation error, the given range is thus the range of both floating-point and real value of the variable.

You can first declare that a variable `x` has a bottom (empty) value. It can be used if the user knows or wants the analyser to know for sure that in some branch of execution, `x` is not assigned. It can also force the analyser (if this is done for all variables known at this location of the program) to consider the sequel of the program as dead code. This assertion takes the type of the returned variable as a parameter.

```
double x;
int i;
char c;
.....
x = BOTTOM(double);
i = BOTTOM(int);
c = BOTTOM(char);
.....
```

On the other side of the spectrum, one can declare :

```
double x;
int i;
char c;
.....
x = TOP(double);
i = TOP(int);
c = TOP(char);
.....
```

This assigns abstract value "top" ($\top$) to variables `x`, `i` and `c`, meaning that these variables can take all possible values defined in the range of their type.

Now, if we want to declare a set of possible input values instead of one value for a variable, we can use :

```
float x;
.....
x=__BUILTIN_DAED_FBETWEEN(a,b);
y=__BUILTIN_DAED_DBETWEEN(c,d);
i=__BUILTIN_DAED_IBETWEEN(e,f);
.....
```

This tells the analyser that, respectively, floating-point variable `x` is between `a` and `b`; double precision variable `y` is between `c` and `d` and finally, integer value `i` is within `e` and `f`. All these variables have no error.

An assertion exists to specify that a constant has no error :

```
float x;
x= FEXACT(0.1);
double y;
y = DEXACT(0.1);
```

In this case, the analyzer takes for x and y, respectively the float or double representation of 0.1 (slightly different from the decimal number 0.1), as the intended machine and real values.

We can define the successors or the predecessors of a simple (resp double) precision floating-point variable using assertions `FSUCC` (resp `DSUCC`) and `FPREC` (resp `DPREC`) :

```
double x1, x2;
...
x1 = DBETWEEN(-DBL_MAX,DPREC(0));
x2 = DBETWEEN(DSUCC(0),DBL_MAX);
```

In the general case, they define the successor or predecessor for normalized floats, execpt for the successor or predecessor of zero, which are the next/previous denormalized float.

The `DPREC` assertion selects the floating-point value below the argument (in this example, the first negative floating-point value below 0). In the same way, `DSUCC` selects the floating-point value above the argument. These assertions are particularly useful when you want to cover specific ranges without intersections between them. For example :

```
double x1, x2;

x1 = DBETWEEN(-DBL_MAX,-2);
x2 = DBETWEEN(DSUCC(-2),0);
```

3.2 Specifying ranges on values and errors of variables

Now we want to specify also a representation error (between real and floating-point value) to a variable. We thus have to decide if the value set is the floating-point or the real value.

- In the non-relational analysis mode, the analysis follows the floating-point control flow and deduces the real value from addition of the floating-point value and the error. It is thus more natural to use assertions on the floating-point value and errors.
- In the relational analysis modes, the analysis uses relations on the real value and errors, and deduces the floating-point value from the real value and error, it is thus more natural (and accurate) to use assertions on the real value and errors.

However, all assertions can still be used with both kinds of analyses.

Because real numbers can not always be exactly represented by floating-point numbers, one may want the analyser to understand that only an approximation of a constant is known, and that the error committed by this approximation is bounded by the ulp of this constant :

```
double x;
float y;
.....
x = __BUILTIN_DAED_DBETWEEN_WITH_ULP(a,b);
y = __BUILTIN_DAED_FBETWEEN_WITH_ULP(a,b);
.....
```

Variable `x` has a floating-point value in $[a, b]$, and it may have an initial rounding error bounded by $[-ulp(c), ulp(c)]/2$, where $c = \max(|a|, |b|)$.

One can also specify an initial error together with a range of values for each variable :

```
double x;
float y;
int i;
.....
x = __BUILTIN_DAED_DOUBLE_WITH_ERROR(a,b,c,d);
y = __BUILTIN_DAED_FLOAT_WITH_ERROR(a,b,c,d);
i = __BUILTIN_DAED_INT_WITH_ERROR(a,b,c,d);
.....
```

The analyser now knows that variable `x` has floating-point values between constants `a` and `b` and error (localised at the line at which the assertion has been written) between constants `c` and `d`. The real value of `x` has its value between $a + c$ and $b + d$. For variable `y`, this is the same, except `y` is in double precision. The same can be specified for an integer `i`, with integer bounds `a`, `b`, `c` and `d`.

Instead of the absolute error, a maximum relative error (in absolute value) can also be specified together with the range of the floating-point value :

```
double x;
float y;
.....
x = __BUILTIN_DAED_DOUBLE_WITH_RELERROR(a,b,c);
y = __BUILTIN_DAED_FLOAT_WITH_RELERROR(a,b,c);
.....
```

In the code above, `x` is given a floating-point value within the interval `[a,b]`, with a potential error with is between `-c` and `c` times the maximum absolute magnitude of numbers in this interval of floating-point numbers.

There are two main practical uses of these keywords. First, it is useful in order to specify environments in which to analyse some programs : typically external inputs with some known imprecision errors (see examples `stability1` and `stability2` for instance in Section 4.3.1). This can also be used for simulating by hand a modular analysis. Secondly, it can be used for specifying the imprecision at which some constants have been written in the program.

We can now specify to the analyser that a variable `x` takes its real value between constants `a` and `b`, and an error (localised at the line at which the assertion has been written) in $[-d, -c]$, so that the floating-point value of `x` takes its value between $a + c$ and $b + d$.

```
double x;
float y;
int i;
.....
x = __BUILTIN_DAED_DREAL_WITH_ERROR(a,b,c,d);
y = __BUILTIN_DAED_FREAL_WITH_ERROR(a,b,c,d);
i = __BUILTIN_DAED_IREAL_WITH_ERROR(a,b,c,d);
.....
```

3.3 Limiting ranges of values and errors of variables

One may also want to no longer specify ranges of values and errors of inputs, but limit the ranges of values and errors of existing variables. In the same way as above, there are assertions allowing to limit the ranges of floating-point or real values, and it is more natural and accurate to limit the range of the real value in the case of relational analyses.

The following code specifies to the analyser the fact that the floating-point value range of variable `x2` is obtained by intersecting the range of floating-point value of variable `x1` with the interval $[a, b]$, and its error is the error of variable `x1`. The value range of `x2` may thus not take the full range $[a, b]$. This is done typically for helping the analyser when it is not precise enough.

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DOUBLE(x1,a,b);
y2 = __BUILTIN_DAED_LIMIT_FLOAT(y1,a,b);
i2 = __BUILTIN_DAED_LIMIT_INT(i1,a,b);
.....
```

The same assertions, but for limiting the ranges of the real value of variables write :

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DREAL(x1,a,b);
y2 = __BUILTIN_DAED_LIMIT_FREAL(y1,a,b);
i2 = __BUILTIN_DAED_LIMIT_IREAL(i1,a,b);
.....
```

To limit the ranges of the error on a variable, we can write :

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DERROR(x1,a,b);
y2 = __BUILTIN_DAED_LIMIT_FERROR(y1,a,b);
i2 = __BUILTIN_DAED_LIMIT_IERROR(i1,a,b);
.....
```

Used in the non-relational mode, `x2` takes the same floating-point values as `x1`, and its error is limited by the range $[a, b]$. The real values are in $[\min(x1) + a, \max(x1) + b]$.

Used in a relational mode, `x2` takes the same real values as `x1`, and its error is limited by the range $[-b, -a]$, so that the floating-point values are in $[\min(x1) + a, \max(x1) + b]$.

In the case when the bounds are actually used to limit the existing error, the drawback of the use of this assertion may be that the provenance (in terms of lines in the C source code) of the errors is lost, the range of error is associated to the line of the assertion.

Assertions also exist, that limit both the ranges of floating-point values and errors :

```
double x1, x2;
float y1, y2;
```

```
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DFLOAT_AND_ERROR(x1,a,b,c,d);
y2 = __BUILTIN_DAED_LIMIT_FFLOAT_AND_ERROR(y1,a,b,c,d);
i2 = __BUILTIN_DAED_LIMIT_IFLOAT_AND_ERROR(i1,a,b,c,d);
.....
```

For example, variable `x2` is assigned to a floating-point value equal to that of `x1` intersected with $[a, b]$, and its error is assigned to that of `x1` intersected with $[c, d]$, so that the real value of `x2` lies in a sub-interval of $[a + c, b + d]$.

Finally, assertions that limit the ranges of real values and errors are :

```
double x1, x2;
float y1, y2;
int i1, i2;
.....
x2 = __BUILTIN_DAED_LIMIT_DREAL_AND_ERROR(x1,a,b,c,d);
y2 = __BUILTIN_DAED_LIMIT_FREAL_AND_ERROR(y1,a,b,c,d);
i2 = __BUILTIN_DAED_LIMIT_IREAL_AND_ERROR(i1,a,b,c,d);
.....
```

For example, variable `x2` is assigned to a real value equal to that of `x1` intersected with $[a, b]$, and its error is assigned to that of `x1` intersected with $[-d, -c]$, so that the floating-point value of `x2` lies in a sub-interval of $[a + c, b + d]$. In more detailed example is available in Section 4.2.4.

3.4 Proof assertions

Some assertions can be used to ask the analyzer to check if a variable (or an expression) has its value or error within some bounds. These assertions can be used on the floating-point value, the real value or the error :

```
double x, a, b;
float y, c, d;
int i, e, f;
.....
ASSERT_FLOAT(y,a,b);   // assertions on the floating-point value (for a float/double/int variable)
ASSERT_DOUBLE(x,c,d);
ASSERT_INT(i,e,f);

ASSERT_FREAL(y,a,b);   // assertions  on the real value (for a float/double/int variable)
ASSERT_DREAL(x,c,d);
ASSERT_IREAL(i,e,f);

ASSERT_FERROR(y,a,b);  // assertions on the error (for a float/double/int variable)
ASSERT_DERROR(y,a,b);
ASSERT_IERROR(y,a,b);
.....
```

A window dedicated to the result of these assertions is available at the end of the analysis, summaryzing the assertions proved, possibly violated or proved to be violated.

When the bounds specified in the assertions are known facts, that the user wants the analyzer to use for the rest of the program, they can be used in the following way :

```
y = ASSERT_FLOAT(y,a,b);
```

These assertions always returns the value of its first argument reduced using the bounds : when the returned value is assigned in a variable (not necessarily the same as the argument), this allows the analyzer to consider the assertion as a known fact.

3.5 Assertions on the dynamics of the inputs of a system

In some cases, bounds on the values are not sufficient to describe accurately the behaviour of a system : we thus propose an assertion that allows to bound, in a loop indexed by an integer variable `i`, the variation between two successive values of an input variable `x` :

```
double x;
.....
  for (i=i0 ; i<N ; i++) {
    .....
    x = __BUILTIN_DAED_FGRADIENT(x0min,x0max,gmin(i),gmax(i),xmin,xmax,i,i0);
    .....
  }
.....
```

In this assertion, `i0` is the value of variable `i` at first iteration. The value of `x` at first iteration is in interval $[x0min, x0max]$, the difference between two successive iterates is in the interval $[gmin(i), gmax(i)]$, which bounds may depend on the iterate, and the value of `x` is always bounded by $[xmin, xmax]$. Thus $x(i0) = [x0min, x0max]$, and for all $i \in \{i0, \ldots, N\}$, we have

$$x(i) = (x(i-1) + [gmin(i), gmax(i)]) \bigcap [xmin, xmax].$$

Our relational domain (Section **??**) is specially well adapted to dealing with these relations between iterates in a loop.

3.6 Join and meet assertions

These two keywords allow for union or intersection of values and errors :

```
double x,y,z,t;
.....
x=__BUILTIN_DAED_DJOIN(z,t);
y=__BUILTIN_DAED_DMEET(z,t);
.....
```

These take respectively the union of values and errors of `z` and `t` and put it in `x`, and the intersection of values and errors of `z` and `t` and put it in `y` (in the example here, these assertions are defined for double variables, but there exist equivalents for float or integer variables.

As a practical use, the user can help the analyser (using the intersection operator) or write stubs for some functions, for which we do not have the code or that we do not want to analyse (see for instance Sections 4.2.1 and 4.2.2).

Also, the JOIN assertion can be used for improving the precision of analysis by a local and possibly irregular subdivision, coded in the program to analyse, see Section **??**. In some cases when the result of interest is the variable that is computed using this `JOIN` assertion, and relations between this variable and inputs are not needed for further computations, the assertion `__BUILTIN_DAED_DCOLLECT` can be used instead of `__BUILTIN_DAED_DJOIN`. With `DCOLLECT`, non relational union are made between the different abstract values that are joined, which may be much more efficient in time.

`__BUILTIN_DAED_DCOLLECT_WARN`, `__BUILTIN_DAED_FCOLLECT_WARN` or `__BUILTIN_DAED_ICOLLECT_WARN` do the same action as `__BUILTIN_DAED_DCOLLECT`, `__BUILTIN_DAED_FCOLLECT` or `__BUILTIN_DAED_ICOLLECT`, but in addition issue a warning when there is a hole between values of collected variables (see example in Section 4.5.5).

3.7 Assertions to print information during the analysis

All information displayed by default by the interface, is about the state of variables at the end of the program. If the user wants to visualise information during the analysis, he must specify which variable he wants to see, and at which place in the program.

This is done for a floating-point variable `S` for instance, by inserting an assertion

```
__BUILTIN_DAED_FPRINT(S)
```

at some places in the program. Of course, the corresponding assertions `__BUILTIN_DAED_DPRINT(S)` and `__BUILTIN_DAED_IPRINT(S)` exist for double precision floating-point and integer variables.
Then, the analyser will print in a file, the interval value and global error for this variable, each time it meets the assertion. For example, if it is in a loop, it will print the value at each fixpoint iteration. Some buttons in the user interface then allow to draw figures representing this evolution (see Section 2.5.3).

Also, when a relational mode of the analyser is selected, a more complete representation of the values of variables can be printed using assertion `__BUILTIN_DAED_FAFFPRINT(S)`.

3.8 Assertions to retrieve input-output relations

`DSENSITIVITY(x)`, `FSENSITIVITY(x)` or `ISENSITIVITY(x)` allow to print sensitivity of double, float or integer variable `x` to inputs, using the input/output linearized relations given by the affine forms abstract domain. These assertions have to be used outside loops.

When manual subdivisions are used (see section 3.9), it is useful to retrieve sensitivity information for each subdivision. You can do that with the `DCOLLECT_WORST`, `FCOLLECT_WORST` or `ICOLLECT_WORST` assertions. With these builtins, on top of collecting values as the standard `COLLECT` assertion does, the analysis prints sensitivity information. Note that it also warns the user when there is a hole between collected values, in the same way as the `COLLECT_WARN` assertions do. And when the user tries to reach the "minimal" (or "maximal") worst-case, the builtins select the "min" (or "max") value between the collected worst-cases.

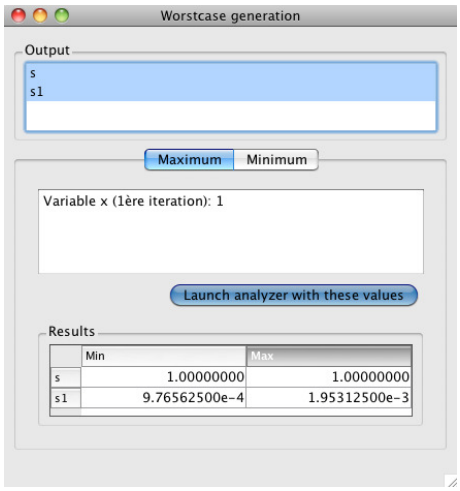
In the following example, we manually subdivide the interval $[1; 1024]$ with a loop. The interval is subdivided as follows :

- $[1; 2]$ for the first iteration,
- $[2; 4]$ for the second,
- ...

For each subdivision, we collect the result `s1` in `s`, and in the same time, we get sensitivity information.

```
double calcul(double a){
  return 1/a;
}

int main(){
  double x;
  double s,s1;
  int i=0;

  while(i<10){
    x = DBETWEEN(1<<i,1<<(i+1));
    s1 = calcul(x);
    s = DCOLLECT_WORST(s,s1);
    i++;
  }
}
```

The example gives in the worst-case generation window (see Section 2.5.10) :



The maximal worst-case generated by Fluctuat is $s = [1; 1]$, which is the maximum between worst-cases of each iteration, and the analyzer signals that this worst-case is obtained for iteration $i = 1$.

3.9 Local and Manual Subdivisions

When global subdivisions on the whole program take a long time, it may be pertinent to replace them with local subdivisions. Local subdivisions apply only on a part of the code specified by the user. Fluctuat provides regular subdivisions, irregular subdivisions, subdivisions to meet a requirement and manual subdivisions. These subdivisions are not parallelisable. The evolution graphs also chain the subdivisions instead of distinguishing them like for the global subdivisions.

3.9.1 Regular Subdivisions

The keywords `SUBDIV_BEGIN` and `SUBDIV_END` define the part of the code to subdivide. The construction

```
double x = DBETWEEN(a, b);
.....
SUBDIV_BEGIN(x, n)
  code
SUBDIV_END
.....
```

where a, b are double constants and where n is an integer constant, subdivides the analysis of `code` in n sub-analysis. Fluctuat first analyses `code` for the subdivision $x \in [a, a + \frac{b-a}{n}]$ and ends the analysis of `code` for the subdivision $x \in [b - \frac{b-a}{n}, b]$.

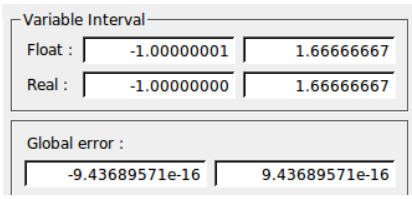
The construction is semantically equivalent to

```
double x = DBETWEEN(a, b);
int subdivide_index;
.....
for (subdivide_index = 0; subdivide_index < n; ++subdivide_index) {
  if (x >= a + subdivide_index*(b-a)/n && x <= a + (subdivide_index+1)*(b-a)/n) {
    code
  }
}
.....
```

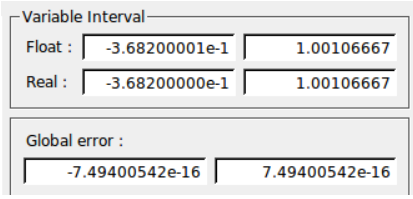
In the last version of the following example, we regulary subdivide the interval $[-2; 2]$ in 100 regular subdivisions.

- $[-2; -1.99]$ for the first iteration,
- $[-1.99; -1.98]$ for the second,
- ...

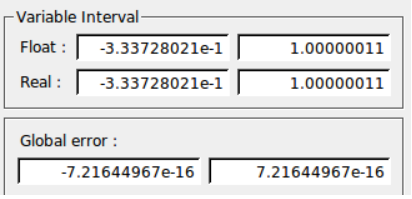
```
double calcul(double a){
  return 1-a*a/2+a*a*a*a/24;
}

int main(){
  double x = DBETWEEN(-2, 2);
  double s1;

  s1 = calcul(x);
}
```




```
double calcul(double a){
  return 1-a*a/2+a*a*a*a/24;
}

int main(){
  double x = DBETWEEN(-2, 2);
  double s1;

  SUBDIV_BEGIN(x, 10)
  s1 = calcul(x);
  SUBDIV_END
}
```




```
double calcul(double a){
  return 1-a*a/2+a*a*a*a/24;
}

int main(){
  double x = DBETWEEN(-2, 2);
  double s1;

  SUBDIV_BEGIN(x, 100)
  s1 = calcul(x);
  SUBDIV_END
}
```



If the variable to subdivide is an integer value, the construction

```
int x = IBETWEEN(a, b);
.....
SUBDIV_BEGIN(x, b-a+1)
  code
SUBDIV_END
.....
```

analyses the code for each enumerated value of x.

3.9.2 Irregular Subdivisions

The keywords `SUBDIV_STEP_BEGIN` and `SUBDIV_END` define the part of the code to subdivide. The construction

```
double x = DBETWEEN(a, b);
double step = ...;
.....
SUBDIV_STEP_BEGIN(x, step)
  code
  step = ...
SUBDIV_END
.....
```

where a, b are double constants, subdivides the analysis of `code` in different sub-analyses whose subdivision's width is *step* for the variable `x`. Fluctuat first analyses `code` for the subdivision $x \in [a, a+step_0]$ and ends the analysis of `code` with the subdivision $x \in [a+\sum_{i=0}^{n} step_i, b]$. n is the first integer such that $a+\sum_{i=0}^{n} step_i \leq b$ and $a + \sum_{i=0}^{n+1} step_i > b$. Each $step_i$ should be strictly positive.

The construction is semantically equivalent to

```
double x = DBETWEEN(a, b);
double x_min;
double step = ...;
.....
for (x_min = a; x_min < b; x_min = x_min + step) {
  if (x >= x_min && x <= x_min + step) {
    code
    step = ...
  }
}
.....
```

In the following example, we irregulary subdivide the interval $[1; 1024]$. The interval is subdivided as follow :

- $[1; 2]$ for the first iteration,
- $[2; 4]$ for the second,
- ...

```
double calcul(double a){
  return 1/a;
}

int main(){
  double x = DBETWEEN(1, 1024);
  double s1, step=1;

  SUBDIV_STEP_BEGIN(x, step)
  s1 = calcul(x);
  step *= 2;
  SUBDIV_END
}
```

Irregular subdivisions are especially interesting when the worst case results are located on few regular subdivisions. They enable to focus the analysis on these subdivisions while keeping large ranges for the other subdivisions. If adequate, this mechanism improves both the analysis' time and the accuracy of the results.

3.9.3 Subdivisions to Meet a Requirement

The keywords `SUBDIV_REQUIRE_BEGIN` and `SUBDIV_END` bound the part of the code to subdivide. The construction

```
double x = DBETWEEN(a, b);
.....
SUBDIV_REQUIRE_BEGIN(x, result, err_min, err_max)
  code
SUBDIV_END
.....
```

where *a*, *b*, *err_ min*, *err_ max* are double constants, subdivides the analysis of code in different sub-analyses whose subdivision's width is an adequate *step*. By adequate, we mean *step* should be such that at the end of the subdivision the global error of the expression `result` remains in the interval [*err_ min*, *err_ max*]. If it is not the case for the current attempt, Fluctuat decreases *step*. If the global error of `result` is only a small part of [*err_ min*, *err_ max*], then Fluctuat increases *step*.

This construction automatically finds subdivisions close to the optimum. Such subdivisions can be exported in terms of irregular subdivisions.

3.9.4 Manual Subdivisions

The manual subdivisions proceed by collecting the outputs. They are close to irregular subdivisions and they enable a finer control by specifying the input instead of limiting it.

For an initial code to be subdivided

```
double x = DBETWEEN(a, b);
double result1;
double result2;
.....
code
result1 = ...
result2 = ...
}
.....
```

a manual subdivision translates it into

```
double x_min;
double result1, result1_temp;
double result2, result2_temp;
double step = ...;
.....
for (x_min = a; x_min < b; x_min = x_min + step) {
  double x = DBETWEEN(x_min, x_min + step);
  code
  result1_temp = ...
  result1 = DCOLLECT_WORST(result1, result1_temp);
  result2_temp = ...
  result2 = DCOLLECT_WORST(result2, result2_temp);
  step = ...
}
.....
```

whereas irregular subdivisions rather means

```
double x_min;
double x = DBETWEEN(a, b);
double result1;
double result2;
double step = ...;
.....
for (x_min = a; x_min < b; x_min = x_min + step) {
```

```
  if (x >= x_min && x <= x_min + step) {
    code
    result1 = ...
    result2 = ...
    step = ...
  }
}
.....
```

This alternative construction respects the sensitivity analysis and the worst case generation.

4 Examples

The following examples are useful for two different purposes. Obviously, they help to understand how the tool works on simple examples, for which it is not too hard to guess the results. They are also of main interest to validate the tool : it is indeed difficult to validate the results provided by the analyser for large programs, since there is no way to compare them to ideal results (calculation carried out with real numbers), or to other tools. The academic examples help us check that every kind of instability is correctly detected by Fluctuat, as predicted by the theory.
The distribution includes the `DEMO` directory who contains some of these so-called academic examples.

4.1 Simple examples

In the `DEMO/SIMPLE` directory, you can find very simple examples to understand the basic concepts of the floating-point computations. This section details some others examples, see the following code :

```
int main()
{
  double x;
  x = DOUBLE_WITH_ERROR(1,2,-0.01,0.02);
}
```

and click on the variable `x` : there is a bar (with maximum value 0.02) in the corresponding histogram. When you click on it, you see the following line selected on the source part of the window :

```
x = DOUBLE_WITH_ERROR(1,2,-0.01,0.02);
```

telling you that this error comes from the computation at this line.

Now you can also try programs like the following one, which contains calls to external "standard libraries", such as some functions from `math.h` :

```
#include <math.h>

int main(void)
{
  float x = 2.999982;
  float a = 0.4;
  float b = 0.8;
  float c, d, e;

  a = FBETWEEN(0,1);
  c = sqrt(a + b);
  e = c - 2*b;
  d = fabs(e);
  if (x > a + b)
     x = 0;
}
```

In case the analysed program does not terminate, the list of variables available at the end of the program contains only `main`, with interval value bottom ($[+\infty, -\infty]$) for value and errors. A warning is also generated.

```
int main(void)
{
```

```
  int i;
  while (1)
    i++;
}
```

4.2 Use of assertions

4.2.1 Helping the analyser

If you erase the `BUILTIN` constructs at lines [a] and [b] in the following code, this will result in a potentially infinite error for `t` coming from line [0] `y=y+x`. As a matter of fact, if the error on `t` was continuously equal to 0.001 and its floating point value would keep on being equal to 0, then the "real" result would be 10 (in fact, in other situations, it could go as far as a difference of 20), whereas the floating-point result would be zero ; the analyser extrapolates this imprecision error and pretends it may be infinite.

In the following code, at line [a] we specify that the error committed for variable `z` at this specific line cannot exceed the interval [-20.0,20.0]. Then we use this extra variable `z` to limit the error on variable `y` so that the error coming from line [a] is truncated for it does not exceed 20 in absolute value.

```
   float y,z,t;

   void integre(float x) {
[0]  y = y+x;
     if (y>10.0)
       y = 10.0;
     if (y<-10.0)
       y = -10.0;
[a]  LIMIT_ERROR(y,z);
   }

   int main() {
     int i;
     y = 0.0;
[b]  t = FLOAT_WITH_ERROR(0,1,-0.001,0.001); \
     z = FERROR_MAX(-20.0,20.0);
     for (i=1;i<10000;i++)
       integre(t);
   }
```

Notice the warnings at the console, indicating that the `if` statement in function `integre` may be unstable.

4.2.2 Creating stubs

In the following program, we define two "stubs" for a special absolute value kind of function `new_abs` and for a home made square root function `home_made_sqrt`. Our absolute value function is abstracted by the sup of its argument with its argument plus one. For the square root function, we first assert that the argument should be positive, by `LIMIT(t,0,HUGE_VAL);` [5]. Then the square root is approximated by floating-point interval $[0, 1]$ when the argument is less than 1, with error between -0.0000001 and 0.0000001. Finally, for an argument greater or equal to one, we use the fact that the square root is below its argument and is greater or equal to one. We add up an error of magnitude at most 0.0000001.

```
float new_abs(float t) {
  float res;
```

5. We have included `math.h` so as to have access to this high value `HUGE_VAL`.

```
  res = FJOIN(t,-t+1);
  return res;
}

float home_made_sqrt(float t) {
  float res;
  LIMIT(t,0,1000);
  if (t < 1.0)
    res = FLOAT_WITH_ERROR(0.0,1.0,-0.0000001,0.0000001);
  else
    res = JOIN(FBETWEEN(1.0,1.0),
                              FEPSILON(-0.0000001,0.0000001)+t);
  return res;
}

int main() {
  float x,y;
  x = FLOAT_WITH_ERROR(-100.0,100.0,-0.001,0.001);
  if (x > 10.0)
    x=15.0;
  if (x < -2)
    x=0;
  x = new_abs(x);
  y = home_made_sqrt(x);
}
```

4.2.3 Specifying constants

The following program shows how to define constants in three different manners :

```
int main() {
  float x,y,z;
  x = .3333333;
  y = FEXACT(.3333333);
  z = FLOAT_WITH_ERROR(.3333333,.3333333,.0000000001,.0000000001);
}
```

- an error may be associated to constant `x` if .3333333 is not exactly representable in float ; its real value is exactly .3333333, and its float value the rounded float.
- constant `y` is considered exact, meaning that the real number corresponding to it is exactly the floating point number representing the decimal string .3333333 (see the value of the variable `x`).
- constant `z` is declared with a specified error of .0000000001.

Finally, the assertion

```
double x;
x = DBETWEEN_WITH_ULP(a,a);
```

specifies that variable `x` has a floating-point value equal to the floating-point representation of `a`, and it may have an initial rounding error bounded by $[-ulp(a), ulp(a)]/2$.

4.2.4 Limit values and errors

```
 double y1,y2,y3,y4,y5,y6;
```

```
y1 = DOUBLE_WITH_ERROR(-400,100,-10,0.1);
y2 = LIMIT_DREAL(y1,0,50);
y3 = LIMIT_DREAL_AND_ERROR(y1,0,50,-100.0,0.0);
y4 = LIMIT_DOUBLE(y1,0,50);
y5 = LIMIT_DOUBLE_AND_ERROR(y1,0,50,-100.0,0.0);
y6 = LIMIT_DERROR(y1,-100.0,0.0);
```

After this sequence of assertions, the analyser prints via the graphic interface the results summed-up in the following table (in which, when the results are the same for relational and non-relational analyses, the type of analysis is not specified) :

variable	real value	floating-point value	error
y1	[-410,100.100001]	[-400,100]	[-10,0.1]
y2 (non-relational)	[-10.1000001,60.1000001]	[-0.1,60]	[-10,0.1]
y2 (fully relational)	[0,50]	[-0.1,60]	[-10,0.1]
y3 (non-relational)	[-0.100000001,50.1000001]	[-0.1,50]	[0,0.1]
y3 (fully relational)	[0,50]	[-0.1,50]	[0,0.1]
y4	[-10,50.1]	[0,50]	[-10,0.1]
y5	[-10,50]	[0,50]	[-10,0]
y6 (non-relational)	[-410,100]	[-400,100]	[-10,0]
y6 (fully relational)	[-410,100.100001]	[-400,100]	[0,0.1]

Indeed, `y2` keeps the same error as `y1` in $[-10, 0.1]$, and its real value is limited in $[0, 50]$. The floating-point value is then obtained by $([0, 50] - [-10, 0.1]) \cap [-400, 100] = [-0.1, 60]$. However, it is the logic of the non-relational analysis that the real value printed is the sum of the range of the floating-point value and the error, thus finally giving as real range $[-0.1, 60] + [-10, 0.1] = [-10.1, 60.1]$. Whereas in the fully-relational mode, the real value printed is the actual one. This shows why the assertions limiting the real values of variables should be avoided in the non relational mode.
In the same way, the real range of `y3` is first truncated to $[0, 50]$, the error is limited by $[0, 100]$, which gives $[0, 0.1]$, and the floating-point value in non relational mode is got by $[0, 50] - [0, 0.1] = [-0.1, 50]$. Same as before, we then print not the actual real range, but that obtained by addition of floating-point range and error, which is $[-0.1, 50.1]$.
When the assertion is set on the floating-point value, the result of the assertion is the same in relational or non-relational mode (see `y4` or `y5`). In the assertion limiting the floating-point value and error of `y5`, note that the error is here limited by $[-100, 0]$.
When we limit only the error, in the non-relational mode, we keep for the floating-point value of `y6` that of `y1`, $[-400, 100]$, and limit the error by $[-10, 0.1] \cup [-100, 0] = [-10, 0]$. This gives a real value in [-410,100]. In the relational mode, we keep the real value of `y1`, $[-410, 100.1]$, and limit the error by $[-10, 0.1] \cup [0, 100] = [0, 0.1]$. The result if of poorer quality than the non-relational one, because the values and errors were set on `y1` with an assertion on floating-point value and error, which is better suited to the non-relational mode.

4.3 Play with parameters

4.3.1 Fixpoint computations in loops

The following example explains the least fixpoint computation in the presence of classical widening/narrowing operators :

```
(1): int i=1;
(2): while (i<100)
       (3): i++; (4):
(5):
```

The abstract semantic equation for the invariant at control point (3) (beginning of the body of the loop) is :

$$I_3 = ([1,1] \cup (I_3 + [1,1])) \cap]-\infty, 99]$$

Without any widening, we get the following fixpoint iteration sequence : $I_3^0 = \bot$, $I_3^1 = [1,1]$, $I_3^2 = [1,2]$,$\cdots$, $I_3^j = [1,j]$, $\cdots$, $I_3^{100} = I_3^{99} = [1,99]$. With a very classical widening operator, after $j << 100$ steps, we get $I_3^{j+1} = [1, +oo[$. Then, $I_3^{j+2} = [1,99]$ (no need for narrowing here !).

Stability (order 1)

```
int main()
{
  float x,y,z;
  int i;
  x=1.0;
  z=1.0;
[a]  y=.618034;
  for (i=1;i<=20;i++) {
    z=x;
    x=y;
[b]    y=z-y; }
}
```

Try this program with parameters :
- standard parameters first (no unfolding, `Widening Threshold` equal to 20)
- complete unfolding (`Cyclic unfolding` greater or equal to $11 = 20/2 + 1$)

In the first case, for variable `z` you see two potentially infinite imprecision errors, one coming from [a], the other from [b]. In the second case, you see that in fact, everything comes from [a] : the original imprecision error is magnified by the loop, in such a way that it becomes of the same order of magnitude as the real result just after 20 loops. The linear iterative scheme is unstable (just check the eigenvalues !).

```
main()
{
  float t,y;
  int i;
  t=1.0;
[a]  y=.618034;
  for (i=1;i<=20;i++) {
[b]    t=t*y; }
}
```

Now try this other program which in theory should compute the same result as the first program, i.e. the gold number power 21. Just try with standard parameters : you will see two infinite imprecision errors for variable `t` coming from [a] and [b]. Now ask for unfoldloopnumber equal to 5 (or more) and let wideningthreshold be equal to 20, then you will see that the imprecision error is negligible (this is stable). In fact you can replace the number of iterations for the loop in the program by any number, even any variable (equal to $\top$) ; the analyser will prove that the error is always negligible.

However in this case, the analyse still requires a certain amount of computations to get a finite estimation of the errors. And if we look more precisely at the influence of the `Cyclic unfolding` and `Widening Threshold` parameters, we notice that in all cases, a bounded estimation of the errors is obtained if and only if the product `Cyclic unfolding * Widening Threshold` is greater than a number which is around 95 to 105, depending on the value of `Cylcic unfolding`. This product represents the number of time iterations of the loop are computed before using widening, so that in all these cases, convergence is not due to the

unfolding of loops, but to the fact that the computed growth of the errors induces no longer a growth of the error intervals (convergence due to the use of finite precision numbers for the analysis).

This rather disappointing behaviour when loops are unfolded, is in this example due to the fact that the value of `y` when entering the loop is a single value and not an interval, so that errors are computed exactly and their behaviour is chaotic, they do not necessarily decrease as would $ulp(y)$. If we substitute "y = FBETWEEN(0,.618034) ;" for "y=.618034 ;" in the second example, then the convergence for the errors is much more easily reached. If we unfold the loop at least twice (`Cyclic unfolding` greater or equal to 2), a bounded value for the error is got for any value of `Widening Threshold`. The bounds for the error obtained, when finite, for same `Cyclic unfolding`, may be a little smaller when `y` is a value and not an interval, but they are still comparable. And the computing time starting from an interval for `y`, and unfolding the loop at least twice, is on this example about 10 times smaller than starting from a value (whatever the value of `Cyclic unfolding`), because the fixpoint is got with only a few iterations.

The parameter `Exact error calculation for constants` can also be used to overcome this problem : setting it to false (errors computed exactly only outside loops) on second example, the analysis converges when unfolding at least 5 times, with any value of `Widening Threshold`.

We can also notice that first order errors converge whenever the loop is unfolded at least 3 times, and only higher order errors, which are in this case negligible, delay the convergence. Setting the parameter `Convergence of higher order errors` to true, we then get the convergence unfolding only three times, but higher order errors are over-approximated.

Finally, executing 4 times (or more) the loop, with parameter `Initial unfolding` set to 1, before beginning the unions over iterations, allows to obtain convergence without unfolding.

Now look at this other linear iterative scheme whatever the values of the parameters, you see that the initial error is not amplified (this scheme is stable for a wide class of input), and with `Cyclic unfolding` at least equal to 3, all errors are bounded.

```
int main(int n) {
  float x,y,z;
  int i;
  x=1.0;
  y = FLOAT_WITH_ERROR(-2.0,3.0,-0.01,0.01);
  y = y-1.0/3.0;
  for (i=1;i<=n;i++) {
    z=x;
    x=y;
    y=(x+z)/6.0; }
}
```

For the following unstable one, the analyser gives infinite bounds for the errors when we do not unfold completely the loop. These warnings are correct : see how the initial error can propagate – up to 10^{15}, the floating-point value may be completely meaningless ! – when the loop is completely unfolded, i.e., when `Cyclic unfolding` is set to a value greater or equal to $51 = 100/2 + 1$.

```
int main()
{
  float x,y,z;
  int i;
  x=1.0;
  z=1.0;
  y=FLOAT_WITH_ERROR(0,.618034,-0.00001,0.00001);
  for (i=1;i<=100;i++) {
    z=x;
    x=y;
    y=z-y; }
```

```
}
```

Now let us look again at linear iterative schemes, but this time for solving a simple set of linear equations by Jacobi's method :

$$\left\{ \begin{array}{rcl} 2x+y & = & \frac{5}{3} \\ x+3y & = & \frac{5}{2} \end{array} \right.$$

which solution is $x = \frac{1}{2}$ and $y = \frac{2}{3}$. The first method below uses the recurrence scheme :

$$\left\{ \begin{array}{rcl} x_{n+1} & = & \frac{5}{6} - \frac{1}{2}y_n \\ y_{n+1} & = & \frac{5}{6} - \frac{1}{3}x_n \end{array} \right.$$

We want to see if, when we modify slightly the coefficients of this recurrence scheme, and when we have an uncertainty on the initial values, do we still get sensible results ?

This is the following program :

```
int main () {

  double x1,y1,x2,y2;
  double a,b,c,d;
  int i;

  a = DOUBLE_WITH_ERROR(0.8,0.85,0.1,0.1);
  b = DOUBLE_WITH_ERROR(0.4,0.6,0.1,0.1);
  c = DOUBLE_WITH_ERROR(0.8,0.85,0.1,0.1);
  d = DOUBLE_WITH_ERROR(0.3,0.35,0.1,0.1);

  x2 = DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);
  y2 = DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);

  for(i=0;i<100000;i++) {
    x1=x2;
    y1=y2;
    x2 = a-b*y1;
    y2 = c-d*x1;
  };
}
```

This can be shown to be stable by the analyser. Either with `Cyclic unfolding` lower or equal to 20 and `Cyclic unfolding * Widening Threshold` greater than some number between 54 and 84 that depends on the value of `Cyclic unfolding`. Or with `Cyclic unfolding` greater or equal to 21 and any value of `Widening Threshold`. When `Cyclic unfolding` increases between two experiments, the bounds of the intervals usually get tighter.

Let us look now at Jacobi's method again, for a linear system

$$\left\{ \begin{array}{rcl} 2x+3y & = & 3 \\ 4x+\frac{3}{2}y & = & 3 \end{array} \right.$$

which has the same solutions, but for which Jacobi's iterative method is now :

$$\left\{ \begin{array}{rcl} x_{n+1} & = & \frac{3}{2} - \frac{3}{2}y_n \\ y_{n+1} & = & 2 - \frac{8}{3}x_n \end{array} \right.$$

Same experiment as before : we perturb coefficients and initial data ; this is the program :

```
int main () {

  double x1,y1,x2,y2;
  double a,b,c,d;
  int i;

  a = DOUBLE_WITH_ERROR(1.2,1.6,0.1,0.1);
  b = DOUBLE_WITH_ERROR(1.2,1.6,0.1,0.1);
  c = DOUBLE_WITH_ERROR(2.0,2.5,0.1,0.1);
  d = DOUBLE_WITH_ERROR(2.5,3.0,0.1,0.1);

  x2 = DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);
  y2 = DOUBLE_WITH_ERROR(2.0,3.0,0.1,0.1);

  for(i=0;i<100000;i++) {
    x1=x2;
    y1=y2;
    x2 = a-b*y1;
    y2 = c-d*x1;
  };
}
```

It is found to be unstable by the analyser, which it is indeed.

Now let us try to solve a system of differential equations by Euler's method. Try first the following program :

```
int main () {

  double x1,y1,x2,y2;
  double h;
  int i;

  h = 0.1;
  x2 = DOUBLE_WITH_ERROR(0.99,1.01,0.0,0.1);
  y2 = DOUBLE_WITH_ERROR(1.99,2.01,0.0,0.1);

  for(i=0;i<100000;i++) {
    x1 = x2;
    y1 = y2;
    x2 = (1-2*h)*x1;
    y2 = (1+3*h)*y1 + h*x1;
  };
}
```

Notice that we have initial errors on `x2` and `y2`. The analyser is able to detect that this scheme is stable for `x2` and that it is unstable for `y2` (set `Cyclic unfolding` to at least 9).

Stability (higher-order)

```
int main (int n) {

  double x;
  int i;
```

```
  x = DOUBLE_WITH_ERROR(0.0,0.95,0.0,0.01);

  for(i=0;i<n;i++) {
    x = x*x;
  };
}
```

The previous example shows a stable non-linear scheme. Stability is proven as soon as you choose `Cyclic unfolding` at least equal to 5.

Now the following similar non-linear scheme is stable at order one but not in higher order (take also `Cyclic unfolding` at least equal to 5). In fact, for values of `x` greater or equal to 1, this diverges indeed! This is the program :

```
int main (int n) {

  double x;
  int i;

  x = DOUBLE_WITH_ERROR(0.0,0.95,0.0,0.1);

  for(i=0;i<n;i++) {
    x = x*x;
  };
}
```

Fixpoints of non-linear dynamical systems

```
int main(void)
{
  float x0, x1, x2;
  int i;

  x0 = 11/2.0;
  x1 = 61/11.0;

  for (i=1 ; i<=13 ; i++)
  {
    x2 = 111 - (1130 - 3000/x0) / x1;
    x0 = x1;
    x1 = x2;
   }
}
```

The previous example is a classical example by W. Kahan and J.-M. Muller. If computed with exact real arithmetic, this would converge to 6. If computed with any approximation, this converges towards 100.

Take `Cyclic unfolding` equal to 13, so that the loop is completely unfolded. Click on the button on the interface to see the estimated "real" value with respect to the estimated floating-point value, you will find approximations of 100 and of 6 respectively.

```
x0 : floating-point value [1.000000e2 1.000000e2]
     real value [5.84741865837139 5.97473790637248]
x1 : floating-point value [1.000000e2 1.000000e2]
     real value [4.67444955112023 7.03896201757234]
```

The estimation for the "real" value starts diverging. And it will be worse if we increase the number of iterations of the loop (and accordingly the unfoldloopnumber). Notice that the higher order error is the one that diverges.

We then can use the resource about the number of bits used in the analyser to encode real numbers (`MPFR Bits`), in order to have more precise results for a larger number of iterations. The default number of bits is 60, set it to 65 for example : a much tighter estimate of the real result (and of the error committed) is obtained.

```
x0 : floating-point value [1.000000e2 1.000000e2]
     real value [5.91247078405563 5.91644923891061]
x1 : floating-point value [1.000000e2 1.000000e2]
     real value [5.88973283673352 5.96352951477832]
```

Progressive widening operators for errors :

We have seen in Section 4.3.1 how to use some parameters of the analysis (such as the number of initial executions or the number of unfolding in the loops) in order to get better fixpoint computation. We have also seen in Section 4.3.1 how a higher number of bits for the numbers used for the analysis could be useful to have tighter approximations for some subtle (and unstable) algorithms.

On the contrary, it can also be useful to reduce the number of bits used for the approximation to improve the of convergence of fixpoint computations in loops. For example, take program `gold2` already studied in Section 4.3.1. Suppose we do not make any execution of the loop before beginning the unions and we do not unfold the loop. Then the sequence of errors, if computed exactly, in real numbers, is ever increasing (to a finite limit). However, the computed sequence reaches a fixpoint after a finite number of precision, thanks to the use of finite precision numbers. Thus, with the default number of bits 60 for computing errors, 95 iterations are needed (we need the `Widening Threshold` parameter greater than 95 to get bounds for the error). Whereas with 30 bits of precision, only 50 iterations are needed to get the same result. Indeed, as the variables here are simple precision floating-point variables, 30 bits are highly enough to analyse the behaviour of errors.

On the basis of this idea, we propose widening operators for the computation of errors, more adequate to subtle numerical computations than jumping to fixed steps. On the same program `gold2`, we now keep the default precision of the analysis (`MPFR Bits`) equal to 60, so that double precision computations in the program can be analysed with all the accuracy expected. But, enable the `Progressive Widening` parameter, we decide that this precision will be progressively degraded in loops, after the `Widening Threshold` is reached, and we still do not have a fixed point. Of course, it is mostly interesting to use this progressive widening with a very low value of `Widening Threshold`. Another threshold, `Threshold before standard widening`, sets the maximum number of iterations before widening towards infinity (classical widening). We also have two more parameters, `Minimal precision` that determines the minimum precision under which we do not want to degrade the precision any longer, and `Loss of precision`, that specifies of how many bits of mantissa we degrade the precision at each widening step.

Thus, activating for example `Progressive Widening`, and setting `Widening Threshold` to 2, `Threshold before standard widening` ≥ 32, `Minimal precision` to 20 and `Loss of precision` to 2, we find the same result as with the classical union + widening towards infinity, with only 37 iterations instead of 99 without progressive widening and `Widening Threshold` to 95.

Another advantage is that the progressive widening also uses the classical widening towards zero when values decrease towards zero. Thus, as soon as the computation of errors have converged, the fixpoint is got. And in the case of `gold2` for example, the number of iteration is independent of the value of `Threshold before standard widening`, provided that it is large enough. Whereas when we use the classical widening, we must iterate enough (95) for the errors to converge, but then the fixpoint is still not reached, it is reached only when the widening is used (for the value to converge towards zero). Thus, if the user chooses the `Widening Threshold` too small, the results are imprecise, and if too large, there are too many iterations. And finding the optimal value of `Widening Threshold` is not at all intuitive.

Widening operators for integer variables

```
int main(void) {
  float y;
  for (int i=0 ; i<=100 ; i++)
   y = y+1;
}
```

The widening operator for integer variables had to be adapted to be used together with this modulo representation. Indeed, consider a limited loop on an integer counter, such as the previous program.

With the classical widening to the maximum integer value after some number of iterations, at the iteration following the widening, the maximum integer value loops to the minimum, and an error is committed : we get after narrowing for i in the loop, $[m, 100]$ with a possibly infinite error (instead of $[0, 100]$ with no error). Whereas if using fixed steps in the widening, such as powers of 2 or 10, we use the information that i remains finite and does not loop to its minimum possible value m, and thus no error is committed.

4.3.2 Call stack depth

```
float f(float x)
{  ...
   (1): y=g(x);
   ...
   (2): z=g(2*x);
   ...  }

float g(float z)
{  ...
   (3): t=h(z);
   ...
   (4): u=h(z/2);
   ...  }
```

For instance, in the previous code :
- there are 4 instances of `h` : (1,3), (1,4), (2,3) and (2,4),
- (1,3) and (2,3) are "unioned" (identified) if the call stack depth is 1,
- the 4 instances are analysed separately if the call stack depth is greater or equal to 2.

4.3.3 Subdivide bitwise operations

:

It is possible to approximate quite accurately the result of a bitwise or shift operation on interval operands. But it is more problematic to propagate existing errors without over-approximation. The error coming from errors on the operands, is estimated by the difference between the result of the operation on the sums machine values plus errors, and the result of the same operation on machine values. This error is associated to the label denoting the bitwise operation (the source of existing errors is not tracked, the user has to look at the sources of errors on the operands). When the value is a point, this can be computed without losing too much precision. But when it is an interval, as we do not handle the dependency between the values, the error is over-approximated. For that, we propose a solution, that can be costly in computing time and memory space needed, but is efficient : we propose to subdivide value intervals for the computation of the errors. Reducing the width of the intervals reduces the explosion problem of the intervals. Even if the two operands have interval values, only one (the one with larger width) is subdivided. This possibility and the maximum number of subdivisions is a parameter of the analysis (used by setting the parameter `Refinement of bitwise operations` in the `Subdivisions` sub-menu of the parameters). Take for example the following program :

```
int main(void) {
  int i,j;
  i = INT_WITH_ERROR(1200129,1380006,-4,4);
  j = i & 0x7FFFF0F0;
}
```

With no subdivision of intervals, we get the value of j in $[1200128, 1376496]$, with an error in $[-180464, 176368]$ in 4 M of memory and 0.02 seconds of analysis time. If subdividing at most 500000 times for example, we get the same interval for values, with an error in $[-4096, 4096]$, in 14 M of memory space and 6.5 seconds of analysis time. In that case, all values in the interval are enumerated. This analysis can be costly for large programs, and we can try compromises : for example, with 50 subdivisions, we get an interval for the error just slightly larger, $[-4336, 4336]$, with 4 M of memory and only 0.02 seconds of analysis time, that is the same needs as for the analysis without subdivisions.

Bitwise operations have a clear meaning only when integers are coded on a fixed number of digits. Thus, when encountering a bitwise or a shift operation, we reduce the interval for the real value, got by the sum of the machine interval and the error, to the values belonging to the considered integer type. For example, if i is an operand of a bitwise operation, and is of an unsigned integer type, if $i = [0, 10] + [-1, 1]\varepsilon$, we take as interval for the real value $i = [0, 11]$.

4.4 Use of C ANSI features

4.4.1 Integers

Conversion between integer types :
Conversions between types are handled, with modular representation. Take this program :

```
int main() {
  unsigned int x;
  int y;
  x = 2*2147483647;
  y = (int) x;}
```

Variable x is 4294967294 with no error. When converted to a signed int, it exceeds the maximum possible value, thus the machine value is -2, with an error compared to a real computation (with infinite capacity of integers) equal to 4294967296. The result for the real computation is thus 4294967294, which is the expected value. We recall that the real result can be seen for example with the user interface, ticking the box close to the place where the interval value for the variable is displayed.

Let m and M be the minimum and maximum values that can take an integer of a given type. Because of the modular computation, we allow two possible abstractions for the set of values that can take an integer represented by a couple of integers (a, b) :

- the interval $[a, b]$ if $m \leq a \leq b \leq M$,
- or the union $[m, b] \cup [a, M]$ if $a > b$.

Indeed, when we have the operation $[a, M]+1$, we want to be able to find the exact result $[a+1, M] \cup [m, m]$. With the interval representation, we would only be able to express that the result lies in the whole integer range $[m, M]$.

However, if an integer lying in a union $[m, b] \cup [a, M]$ has to be converted to a larger integer type or a floating-point type, the best estimate of the converted result is then $[m, M]$, because it is a tighter estimate than $[m', b] \cup [a, M']$ where m' and M' are the limits of the larger type.

Because of this representation, the behaviour in loops can be difficult to understand. For example, take :

```
int main() {
  int x;
  x = 0;
  while (x>=0)
```

```
    x++;   }
```

The floating-point result found at the end of the loop is -2147483648, with a potential error in $[0, +\infty]$. This is indeed the best result that can be found because the loop is iterated until x reaches the maximum value for `int` type, then `x++` makes it loop to the minimum `int` value, -2147483648 : just after this instruction, we get $x = [-2147483648] \cup [1, 2147483647]$. Then the loop ends for $x = -2147483648$. With no limitation for integers, x would have gone to ∞ and the loop would have never ended, so there is a potentially infinite error.

Now, the resembling example where variable `x` is no longer of `int` type but `char` (see `overchar.c`) looks disappointing at first : x is found in $[-128, -1]$ when we expected to find exactly -128, still with an error in $[0, +\infty]$. Indeed, the loop test `x>=0` converts x in `int` type before making the comparison : when the maximum `char` value is reached, we have just before this test, $x = [-128] \cup [1, 127]$. Then the conversion to an `int` (larger type) gives $x = [-128, 127]$, and the negative values after the test are found to be $[-128, -1]$.

Particular case of the conversion `float` towards `int` :

When the converted floating-point exceeds the capacity of the integer type of the result, the conversion is not achieved in the same way as between two integer types, with modular representation. Indeed, the result is not defined by the norm. In this case, the result is the whole range of the integer type, we deduce the error from that.

For example, take :

```
int main() {
   float x;
   int y;
   x = FLOAT_WITH_ERROR(512,2147483649,0,4);
   y = (int) x;
}
```

The analyser finds for the value of y the whole integer range. The minimum error is zero, the maximum error is 2147483649-`MIN_INT`+4.

Moreover, when a floating-point number is converted to an integer, all errors are agglomerated on one error point corresponding to the line of the conversion : indeed, if all errors are small, it would lose too much precision to over-approximate all errors from all sources by an integer, it is better to do it only once on the sum of errors. The drawback is that the source of existing errors is lost : if interested by the error due to a conversion, the user must look at the source of errors on the floating-point variable that was converted.

4.4.2 Complex data structures

A basic alias analysis can be added, setting the `Levels of abstraction` parameter to `Abstract arrays as one location` or `Unravel arrays`, (it should be set to `No pointers` for an analysis without alias, more efficient if the program does really contain no complex data structure). For structs and pointers, there is no difference between the analysis with parameter `Abstract arrays as one location` or `Unravel arrays`, the difference arises for arrays. Here are some simple examples demonstrating the features of alias analysis.

Structs :

Now we can define ANSI C `struct` as in the following example :

```
typedef struct pair {
  float x1;
  float x2;
} twofloats;
```

```
void main() {
  twofloats X;
  X.x1=1.1111111111;
  X.x2=2.2222222222;
}
```

Names that are known to the analyser are "dereference" names : hence one would find in the variables' list `X.x1` and `X.x2`. The different fields of a structure are always distinguished (unlike for arrays with `Levels of abstraction` parameter set to `Abstract arrays as one location`).

Arrays :

We can also consider arrays of values as in the following code :

```
void main() {
  float a[2];
  a[0]=1.111111111111;
  a[1]=2.222222222222;
}
```

If the `Levels of abstraction` parameter is set to `Abstract arrays as one location`, the analyser will only know of the array `a[]` as a whole ; i.e. the result in the variables' list will be the only variable `a[]`, with floating-point value between `1.111111...` and `2.222222...`.

If it is set to `Unravel arrays`, the analyser will distinguish all elements of the array : the analysis is more accurate, and of course more costly.

Pointers and aliases :

Pointers can be considered :

```
void main() {
  float x;
  float *y;
  float z;
  z=1.11111111111;
  y=&z;
  x=*y;
}
```

Variable `y` is not in the list of variables, it is known to be an alias by the analyser. The floating point values known to the analyser are that of `z` and `x`. The analyser finds out that $x = z = 1.11111....$

We can also consider also any complex data structures, such as here, arrays of `struct` elements.

```
struct st {
  float x1;
  float x2;
};

main() {
  struct st tab[10];
  tab[0].x1=2.22;
  tab[0].x2=3.33;
}
```

Relevant entries in the list of variables would be `tab[].x1` and `tab[].x2`.

A more complex example :

Here is now a code which is a typical finite differences scientific code. It is an implicit 1^{st} order finite difference method for solving the one-dimensional heat equation, with initial values discretized on a 12 points line, equal to its coordinate. The boundary conditions are that the extrema have value zero : thus in the following code, we see that the heat decreases on each of the 12 points of the line. This scheme is stable.

Setting the `Levels of abstraction` parameter to `Abstract arrays as one location`, the analyser sees the array as a whole, and the analyser cannot see that the scheme is stable.

Setting it to `Unravel arrays`, and setting `Cyclic unfolding` and `Widening Threshold` to sufficiently high values, for instance to 150 and 50 respectively, the analyser finds the scheme stable (with errors at most of order 10^{-5}).

```
main()
{
  float ut[12];
  float vt[12];
  float alpha;
  int i, j;
  alpha = .2;

  for (i=1;i<11;i++)
    ut[i] = i;
  ut[0] = 0;
  ut[11] = 0;

  for (j=1;j<100;j++) {
    for (i=1;i<11;i++)
      vt[i] = (1-2*alpha)*ut[i]+alpha*(ut[i+1]+ut[i-1]);
    for (i=1;i<11;i++)
      ut[i] = vt[i];
  }
}
```

4.4.3 Unspecified behaviours

The computation of the super-set of possible floating-point values relies on the assumption that we know for sure what operations are executed. First we suppose the order of evaluation of arithmetic expressions (the more natural, left to right, innermost-outermost evaluation), and that the compiler makes no reorganisation. Second we assume that the intermediate computations in arithmetic expressions are rounded according to the specified format (IEEE754, MIL1750, ...), and not in larger registers.

Evaluation order in arithmetic expressions :
Consider on Pentium the following example :

```
int main()
{
  int i;
  float a,b,c,d;
  float x,y;

  a = 90.53;
  b = 15.32;
```

```
  c = 10.78;
  d = 95.07;

  x = 0;
  y = a + b - c - d;
  for (i = 1; i <= 10000 ; i++)
    x += i*y;
}
```

Let us first look at the result of the execution of this simple program with different compilation options :

```
gcc -[/g/O0] evaluation_order1.c
a.out : > x = -4.768648e+01
gcc -[O1/O2/O3] evaluation_order1.c
a.out : > x = 0
```

The different possibilities in the order of evaluation of the three arithmetic operations giving y lead to this dramatic difference on the value of x. This with or without the presence of parentheses to indicate the order of evaluation ; they are not taken into account. Indeed, these operations are associative for a computation in real numbers, but not in floating-point numbers, due to the rounding errors. If computed exactly, y and thus x are equal to zero. But, in simple precision floating-point numbers, rounding errors may be committed on the result of some of these addition and subtractions, and these errors depend on the values of the operands, thus on the evaluation order. This example is designed to deliberately amplify this rounding error, but this situation can very well occur in real case examples, and the user must be aware of the danger. Splitting the computations as

```
y = a+b;
y = y-c;
y = y-d;
```

and

```
y = a+b;
y = y-d;
y = y-c;
```

allows to understand what computations are really executed. However, if using the optimising options of the compilers, even splitting computations using intermediary variables is not enough to ensure what really happens.

Now we consider the analysis. The computation of the 'real' value should not really depend on evaluation order (but as the analysis uses also finite precision numbers, the tightness of the enclosure of the 'real' value depends itself on the evaluation order). But, to be able to compute the super-set of the floating-point value, the analyser has to make an assumption on the evaluation order of the operations : we take for that the natural order, from left to right, and innermost-outermost. This way, the floating-point result is $x = 0$: we can note that it is not the same result as the one given by a compilation with no optimisation option !

Remark : the 'real' value computed for x by the analyser, when unfolding completely the loop for more precision, is $4.44134e-07$, whereas, if computed exactly, y and x are equal to zero. This is due to the fact that constants a, b, c and d are not represented exactly in double precision floating-point numbers, and these rounded values are the values considered as 'real' by the analyser (see for example Section 4.2.3 for details on this current limitation of the analyser).

Storage of intermediary floating-point results in larger registers :
Consider the following example :

```
main () {
   int tab[2]={0xffffffff,0x408c1fff};
   double E1=*(double *)tab; /* 900.0 - ulp(900.0) */
   double E2 = E1+180.0;
   int pe=(E1+180.0)/360.0;
   int pe2=E2/360.0;
}
```

The execution on a Pentium finds `pe=2` and `pe2=3`. Indeed, in the computation of `pe`, the result of `E1+180` is not rounded to the IEEE format (storage in a register), whereas we enforce this rounding by using an intermediary variable in the computation of `pe2`.

4.5 Warnings

4.5.1 Invalid operations

Even if Fluctuat don't manage with run-time errors, it can produce warnings about invalid behavior of programs. Considering the following program :

```
   main(){
      double x = DBETWEEN(-100,100);
      double y,z,t;

(1)   y = sqrt(x);
(2)   z = log(x);
(3)   t = 1/x;
   }
```

For the lines (1) and (2), Fluctuat advertises about invalid inputs for mathematical functions. And in the line (3), it generates a "Division by zero" warning.

4.5.2 Invalid conversions

In some case, it is impossible to convert a value of type t form another value of type t' without overflow. Fluctuat detects it, likes in the following program :

```
#include <float.h>

main(){
   double x = DBETWEEN(-DBL_MAX,DBL_MAX);
   int y = (int) x;
}
```

4.5.3 Programs that never end

```
main(){
   ...
   while(1){
     /* Read inputs */
     ...
     /* Compute */
     ...
     /* Write outputs */
     ...
```

```
   }
}
```

In the previous program, Fluctuat indicates program will never finish.

4.5.4 Unstable tests

We call *unstable test* a test (for example '`if` $(x \leq 10)$ {...}') where, due to the error on the variables involved, the branch taken in a computation achieved with floating-point numbers is not the same branch that would be taken in the same computation with real numbers.
All possibly unstable tests are indicated on the console during the analysis by warning messages specifying the number of the line in the source code where the test occurred. They are also indicated in a file `warnings.xml` But, to compute the errors committed with respect to the computation with real numbers, we make the assumption that the paths followed by the computation with real and floating-point numbers are the same. A discussion of this choice is given at the end of this Section.

Two different messages are possible, depending on whether the test is sure to be unstable, or whether it may or not be unstable. Consider the following example :

```
main() {
  float x, y;
  x = FLOAT_WITH_ERROR(10,10,0.1,0.1);
  if (x <= 10)
    y = x + 1;
  else
    y = x - 1;
}
```

The test 'if (x $\leq$ 10)' is unstable. Indeed, for floating-point values, we have $x \leq 10$, the statement $y = x + 1$; is executed. But for the real value, $x = 10.1 > 10$, the statement $y = x - 1$; would be executed.

Now consider the program :

```
main() {
  float x, y;
  x = FLOAT_WITH_ERROR(10,10,-0.1,0.1);
  if (x <= 10)
    y = x + 1;
  else
    y = x - 1;
}
```

The test 'if (x $\leq$ 10)' is potentially unstable, the two branches are possible for the real value depending on the error : the test is unstable when the error on x is strictly positive, and stable when it is negative or zero.

On such examples as those shown above, the unstable test can be the origin of important differences between the computation in real numbers or in floating-point numbers (the computed value of y in unstable_test1.c is 11 and the value which is considered as the "real" value by the analyser is 11.1, whereas the real value is 9.1), and these differences are not taken into account by the current analysis.
On the contrary, when the outputs of a program containing some tests are continuous with respect to the inputs of the tests, which is often the case in practice, unstable tests are not such a threat. Consider for example :

```
main() {
  float x, y;
  x = FLOAT_WITH_ERROR(10,10,0.1,0.1);
  if (x <= 10)
    y = 2x;
  else
    y = 3x - 10;
}
```

The computed value of y is 20, the value considered as the real value by the analyser is 20.2, and the real value is 20.3. And the smoother the functional around the test, the smaller the error with respect to the real value.
It is thus important that the user looks at all unstable tests indicated, and tries to determine their consequence on the results that follow.

One reason why we make the assumption that the paths followed by the computation with real and floating-point numbers are the same, is the following : consider a test $x < y$ where the floating-point values of x and y are in intervals that have a non empty intersection. If one of the two variables presents an error, even very small, the analyser detects a possibly unstable test. But practically, if the error is very small, the real and floating-point paths will almost always be the same. Thus, taking in account all possible paths in the computation of errors may be a very pessimistic over-approximation, in addition to the fact that it is quite complex to achieve.

It may however be proposed as an option of the analyser in the future, to cover also the path taken by the real value, and when the paths meet again, to take the distance between the estimation of the real value in the two paths as an approximation of the additional error due to the unstable test. The 'main' path remains the floating-point path, and an over-approximation of the error committed by taking the wrong path is added to the classical errors.

4.5.5 Fluctuat specific warnings

If bounds of intervals specified with `BETWEEN` assertions are not exactly represented in floating-point numbers, Fluctuat generates a warning. For example, $x = DBETWEEN(0, 0.1)$; will issued a warning because 0.1 is not exactly represented.

Some warnings can be generated from Fluctuat assertions `DCOLLECT_WARN`, `FCOLLECT_WARN` and `ICOLLECT_WARN`. These warnings are issued when values collected are not contiguous. Contiguity is checked on real and floating-point values. Consider the following example :

```
   int main(void) {
    int i;
    double E0, E, E1, E2;
    for(i=0;i<10;i++){
      E0 = DBETWEEN(i*10,(i+1)*10);
(1)   E2 = DCOLLECT_WARN(E2,E0);
    }
    for(i=0;i<10;i++){
      E0 = DBETWEEN(i*10,DPREC((i+1)*10));
(2)   E1 = DCOLLECT_WARN(E1,E0);
    }
    for(i=0;i<10;i++){
      E0 = DBETWEEN(DSUCC(i*10),DPREC((i+1)*10));
(3)   E = DCOLLECT_WARN(E,E0);
```

```
    }
   }
```

In (1), no warnings will be issued because between two iterations of the loop all the values are taken and the input intervals used in `DCOLLECT` assertion are contiguous.
In (2), we use the `DPREC` assertion to avoid imprecision of the analysis in tests. But even if the floating-point values are all taken, it is not the case for real values because the interval $]DPREC(i * 10); i * 10[$ is missing (in floating-point, this interval is empty). Fluctuat shows this missing interval with a warning.
And finally in (3), two warnings will be issued to point the missing intervals in floats and in reals.

For some mathematical functions like "log", the norm doesn't specify the behavior with floating-point numbers. Fluctuat issued a warning to point the errors it generates for this operation could be different from those of the user mathematical library. A short example of this behavior :

```
#include <math.h>

int main(void){
  double x, y;
  x = DBETWEEN(10,100);
  y = log(x);
}
```

5 Inside Fluctuat

5.1 Introduction

5.2 Static analysis

5.2.1 Interval arithmetic

5.2.2 Affine arithmetic

Affine arithmetic was proposed by Comba, De Figueiredo and Stolfi [4, 20], as a solution to the overestimation in interval arithmetic, for bounding the result of a computation in real numbers. After a short introduction on affine arithmetic, we describe very shortly the domain we proposed in [18, 11] and implemented in Fluctuat, to estimate the floating-point value f^x in expression (4), domain which is based on affine arithmetic.

In affine arithmetic, a quantity x is represented by an affine form, which is a polynomial of degree one in a set of noise terms ε_i :

$$\hat{x} = \alpha_0^x + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n, \text{ with } \varepsilon_i \in [-1, 1] \text{ and } \alpha_i^x \in \mathbb{R}. \quad (1)$$

Each noise symbol ε_i stands for an independent component of the total uncertainty on the quantity x, its value is unknown but bounded in [-1,1] ; the corresponding coefficient α_i^x is a known real value, which gives the magnitude of that component. The same noise symbol can be shared by several quantities, indicating correlations among them. These noise symbols can be used not only for modelling uncertainty in data or parameters, but also uncertainty coming from computation.

The assignment of a variable x whose value is given in a range $[a, b]$, introduces a noise symbol ε_i :

$$\hat{x} = (a + b)/2 + (b - a)/2\, \varepsilon_i.$$

The result of linear operations on affine forms, applying polynomial arithmetic, can easily be interpreted as an affine form. For example, for two affine forms $\hat{x}$ and $\hat{y}$, and a real number r, we get

$$\begin{aligned} \hat{x} + \hat{y} &= (\alpha_0^x + \alpha_0^y) + (\alpha_1^x + \alpha_1^y)\varepsilon_1 + \ldots + (\alpha_n^x + \alpha_n^y)\varepsilon_n \\ \hat{x} + r &= (\alpha_0^x + r) + \alpha_1^x \varepsilon_1 + \ldots + \alpha_n^x \varepsilon_n \\ r\hat{x} &= r\alpha_0^x + r\alpha_1^x \varepsilon_1 + \ldots + r\alpha_n^x \varepsilon_n \end{aligned}$$

For non affine operations, the result applying polynomial arithmetic is not an affine form : we select an approximate linear resulting form, and bounds for the approximation error committed using this approximate form are computed, that create a new noise term added to the linear form.

Using affine arithmetic for the estimation of floating-point values needs some adaptation. Indeed, the correlations that are true on real numbers after an arithmetic operation, are not exactly true on floating-point numbers. Consider for example two independent variables x and y that both take their value in the interval [0,2], and the arithmetic expression $((x + y) - y) - x$. Using affine arithmetic in the classical way, we write $x = 1 + \varepsilon_1$, $y = 1 + \varepsilon_2$, and we get zero as result of the expression. This is the expected result, provided this expression is computed in real numbers. But if we take x as the nearest floating-point value to 0.1, and $y = 2$, then the floating-point result is $-9.685755e - 8$.

In order to model the floating-point computation, a rounding error must thus be added to the affine form resulting from each arithmetic operation. We decompose the floating-point value f^x of a variable x resulting from a trace of operations, in the real value of this trace of operations r^x, plus the sum of errors δ^x accumulated along the computation, $f^x = r^x + \delta^x$, in which r^x is computed using affine arithmetic.

A natural idea for the error δ^x is to add an interval term, or a new noise term for each operation. However, this is not always satisfying. We thus use an error expressed as the combination of a maximum error (interval term or noise symbol), and the errors associated to the minimum and maximum real values. These errors

on the bounds can often be computed more accurately and are the only information needed to compute the bounds of the floating-point value. But the maximum error may still be needed for the computation of further arithmetic operations.

For static analysis, we need to define an order, and meet and join operations over this domain. This is not obvious because there are no natural union and intersection on affine forms. Also, affine arithmetic was defined with real coefficients : in order to have a computable abstraction, we have to consider the errors due to the use of (arbitrary precision) floating-point numbers instead of real numbers. We refer the user to [18, 11] for more precision about these points, but we are still working to improve the first solutions proposed in these references.

5.2.3 Iteration algorithm

When encountering a loop, which body is represented by a function f, the analyser computes an upper approximation of its fixpoint $\mathrm{lfp}(f) = \cup_n f^n(\bot)$ by an iterative algorithm, that solves the equivalent problem of finding an upper approximation for $\lim_n X_n$, where $X_0 = \bot$ and $X_n = X_{n-1} \cup f(X_{n-1})$.

Standard algorithm (with `Cyclic unfolding` $= 1$) : each time the analyser enters the body of the loop, it compares the present values of the variables with those got at the previous iteration :

- if they are not comparable or if the new values are larger than the older, it takes as new values the union of the old and new (after a number of unions defined by the parameter named `Widening Threshold`, a "widening" towards 0 or infinity is made so that we are sure this process ends within a reasonable number of iterations). And the remaining of the body of the loop is traversed with the new values
- if the values of all variables are the same for the present and past iterations of the algorithm, a fix-point is reached, we quit the loop.
- if the new values are included in the older ones, the old values give an approximation of a fix-point, but we try to get a better approximation by "narrowing" the old values using the new ones. The remaining of the body of the loop is traversed, then the analyser quits the loop (only one narrowing).

With `Cyclic unfolding` ≥ 2 :

In many cases, we noticed that with the previous algorithm, we get infinite bounds for variables whereas the program is stable, and there exists some simple finite bounds. This is due to the fact that when at each iteration we union the new values with the older one, we cannot see that for example the value of a variable decreases with the iterations, and that so does the error. Let us consider the following example :

```
t = 1;
for (i=1 ; i<=100 ; i++)
  t = t*0.618;              (epsilon)
```

With the above algorithm, we get :

$i = 1$:

$$t_1 = [0.618, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon$$

$i = 2$:

$$t_2 = t_1 * 0.618 = [0.618^2, 0.618^2] + 2 * 0.618^2\, ulp(1)[-1, 1]\, \epsilon$$

Noticing that $2 * 0.618^2$ is greater than 0.618, the union with the previous iteration writes

$$t_2 := t_2 \cup t_1 = [0.618^2, 0.618] + 2 * 0.618^2\, ulp(1)[-1, 1].$$

Indeed, we can only bound the error committed at each iteration by the ulp of the largest possible value for t, and this error adds to the previous error. As the maximum value for t, with the unions, is constant, the total error increases.

$i = 3$:

$$t_3 = t_2 * 0.618 = [0.618^3, 0.618^2] + (2 * 0.618^3 + 0.618^2)\, ulp(1)[-1, 1]\, \epsilon$$

$$t_3 := t_3 \cup t_2 = [0.618^3, 0.618] + (2 * 0.618^3 + 0.618^2)\, ulp(1)[-1, 1]\, \epsilon$$

The computed error is again increasing for the same reason ($2 * 0.618^3 > 0.618^2$), thus we finally get infinite bounds for the error $(]-\infty, +\infty[)$, whereas the error really committed decreases, because the value of t decreases. In this particular case, we could get finite bounds with a (very) good widening, since the limit of the error given by this algorithm is $\frac{1}{1-.618} ulp(1)$. But there is a better solution, by unfolding twice the loop : let us rewrite the program as

```
t = 1;
for (i=1 ; i<=100 ; i++) {
  t = t*0.618;              (epsilon 1)
  t = t*0.618;              (epsilon 2)
}
```

With the same algorithm applied to this rewritten program, we get :
$i = 1$:

$$\begin{cases} t_1 = [0.618, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon_1 \\ t_2 = t_1 * 0.618 = [0.618^2, 0.618^2] + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_2 \end{cases}$$

We can run two iterations before joining the values, thus the errors are smaller.
$i = 2$:

$$\begin{cases} t_3 = t_2 * 0.618 = [0.618^3, 0.618^3] + 2 * 0.618^3\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^3\, ulp(1)\, [-1, 1]\epsilon_2 \\ t_4 = t_3 * 0.618 = [0.618^4, 0.618^4] + 2 * 0.618^4\, ulp(1)[-1, 1]\epsilon_1 + 2 * 0.618^4\, ulp(1)\, [-1, 1]\epsilon_2 \end{cases}$$

$$\begin{cases} t_3 := t_3 \cup t_1 = [0.618^3, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^3\, ulp(1)\, [-1, 1]\epsilon_2 \\ t_4 = t_4 \cup t_2 = [0.618^4, 0.618^2] + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_2 \end{cases}$$

$i = 3$:

$$\begin{cases} t_5 = t_4 * 0.618 = [0.618^5, 0.618^3] + (0.618^2 + 0.618^3)\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^4\, ulp(1)\, [-1, 1]\epsilon_2 \\ t_6 = t_5 * 0.618 = [0.618^6, 0.618^4] + (0.618^3 + 0.618^4)\, ulp(1)\, [-1, 1]\epsilon_1 + (0.618^5 + 0.618^4)\, ulp(1)\, [-1, 1]\epsilon_2 \end{cases}$$

$$\begin{cases} t_5 := t_5 \cup t_3 = [0.618^3, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^3\, ulp(1)\, [-1, 1]\epsilon_2 \\ t_4 = t_5 \cup t_3 = [0.618^4, 0.618^2] + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_1 + 0.618^2\, ulp(1)\, [-1, 1]\epsilon_2 \end{cases}$$

Convergence is reached for the errors. We get from $t_4 \cup t_5$ that the error on t in the loop is in $(0.618 + 0.618^3)\, ulp(1)\, [-1, 1] \approx 0.854\, ulp(1)\, [-1, 1]$. We still have to iterate on the values to get the result that the floating-point value of t is in $[0, 0.618]$. We intend to ameliorate the widening operations to take into account such phenomena, in a future version of the tool : it is always more interesting to widen integer values first, then the floating-point range of the abstract values for floating-point variables, and then at the end, the imprecision errors.

Notice that we can also use only one same ϵ for the two unfolded lines (and this is what is practically implemented in the analyser). In some cases, it could lead to a quicker convergence for the errors.

This idea explains the existence of the `Cyclic unfolding` parameter : the user can decide whether he wants loops to be unfolded in the analysis, and how many unfoldings are wanted. In the above example, two unfoldings of the loop are enough to get good results, but in some other cases, more unfoldings will be necessary.

Suppose we unfold the loops N times and f is the functional associated with the body of the loop. The union (or widening) operator is applied in a cyclic way. That is, we compute an upper approximation to $\lim_n(X_1^n, X_2^n, \ldots, X_N^n)$ with X_i^n such that

$$\begin{cases} X_i^0 = \bot, \ i = 1 \ldots N, \\ X_i^n = X_i^{n-1} \cup f^{(i)}(X_N^{n-1}), \ i = 1 \ldots N, \ n \geq 1. \end{cases}$$

When the fix-point is found, the analysis goes on after the loop with $X = \bigcup_{i=1}^{N} X_i$.

This unfolding technique can also be used to unfold completely some loops. For instance when the number of iterations is not too big, and we either want a very tight approximation of the values after a stable loop, or the loop is unstable but we do not want to get infinite bounds.

5.3 Semantics

5.3.1 Semantic of floating-point numbers

In this section, we briefly introduce the semantics of floating-point numbers with errors which is detailed in [8, 14]. Let us examine a simple calculation involving two values $a_{\mathbb{F}} = 621.3$ and $b_{\mathbb{F}} = 1.287$. For the sake of simplicity, we assume that $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$ belong to a simplified set of floating-point numbers composed of a mantissa of four digits written in base 10. We assume that initial errors are attached to $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$, and we write $a = 621.3 + 0.05\vec{\varphi}_1$ and $b = 1.287 + 0.0005\vec{\varphi}_2$ to indicate that the value of the initial error on $a_{\mathbb{F}}$ (resp. $b_{\mathbb{F}}$) is 0.05 (resp. 0.0005). Formal variables $\vec{\varphi}_1$ and $\vec{\varphi}_2$ are related to the static control points at which these errors were introduced (either because there is an imprecision on an input, or because of an operation). a and b are called *floating-point numbers with errors*, and represent the result of the same computation as $a_{\mathbb{F}}$ and $b_{\mathbb{F}}$ but using real numbers instead of floating-point numbers.

Let us focus on the product $a_{\mathbb{F}} \times b_{\mathbb{F}}$ whose exact result is $a_{\mathbb{F}} \times b_{\mathbb{F}} = 799.6131$. This calculation carried out with floating-point numbers with errors yields $a \times b = 799.6131 + 0.06435\vec{\varphi}_1 + 0.31065\vec{\varphi}_2 + 0.000025\vec{\varphi}_1\vec{\varphi}_2$ which we rewrite as $a \times b = 799.6131 + 0.06435\vec{\varphi}_1 + 0.31065\vec{\varphi}_2 + 0.000025\vec{\varphi}_{hi}$. The rewriting step is made to keep only one formal variable per coefficient and obeys the following rule. The indices of the formal variables $\vec{\varphi}_1$, $\vec{\varphi}_2$ etc. are viewed as words of the alphabet of the control points and the product of two variables yields a new formal variable by concatenation of the index words. A word of length one describes a first-order error and the special word hi is used to identify all the words of length greater than one.

The difference between $a_{\mathbb{F}} \times b_{\mathbb{F}}$ and 621.35×1.2875 is 0.375025 and this error stems from the fact that the initial error on a (resp. b) was multiplied by b (resp. a) and that a second-order error corresponding to the multiplication of both errors was introduced. So, at the end of the calculation, the contribution to the global error of the initial error on a (resp. b) is 0.06435 (resp. 0.31065) and corresponds to the coefficient attached to the formal variable $\vec{\varphi}_1$ (resp. $\vec{\varphi}_2$). Finally, the number 799.6131 has too many digits to be representable in our floating-point number system and we refer to the IEEE-754 norm for floating-point arithmetic to determine how the values are rounded [3]. Let $\mathbb{F}$ be either the set of simple or double precision floating-point numbers. The norm fully specifies the function $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ which returns the rounded value of a real number $r \in \mathbb{R}$ with respect to the current rounding mode $\circ$ [3, 7]. In addition, it ensures that the elementary operations are correctly rounded, i.e. for any operator $\Diamond \in \{+, -, \times, \div\}$, we have :

$$\forall f_1, f_2 \in \mathbb{F}, \ f_1 \Diamond_{\mathbb{F}} f_2 = \uparrow_\circ (f_1 \Diamond_{\mathbb{R}} f_2) \tag{2}$$

The function $\downarrow_\circ : \mathbb{R} \to \mathbb{F}$ that returns the roundoff error is then defined by

$$\forall x \in \mathbb{R}, \ \downarrow_\circ (x) = x - \uparrow_\circ (x) .$$

In Equation (2), $\Diamond_{\mathbb{F}}$ and $\Diamond_{\mathbb{R}}$ denote the same operation over $\mathbb{F}$ and $\mathbb{R}$. Assuming that the machine we are using conforms to that standard, we may claim that the computed floating-point number for our example is

$\uparrow_\circ (799.6131) = 799.6$ and that a new first error term $0.0131\vec{\varphi}_\times$ is introduced by the multiplication. To sum up, we have

$$a \times b = 799.6 + 0.06435\vec{\varphi}_1 + 0.31065\vec{\varphi}_2 + 0.000025\vec{\varphi}_{hi} + 0.0131\vec{\varphi}_\times \tag{3}$$

At first sight, one would think that the precision of the calculation mainly depends on the initial error on a since it is 100 times larger than the one on b. However the above result indicates that the final error is mainly due to the initial error on b. Hence, to improve the precision of the final result, one should first try to increase the precision on b (whenever possible).

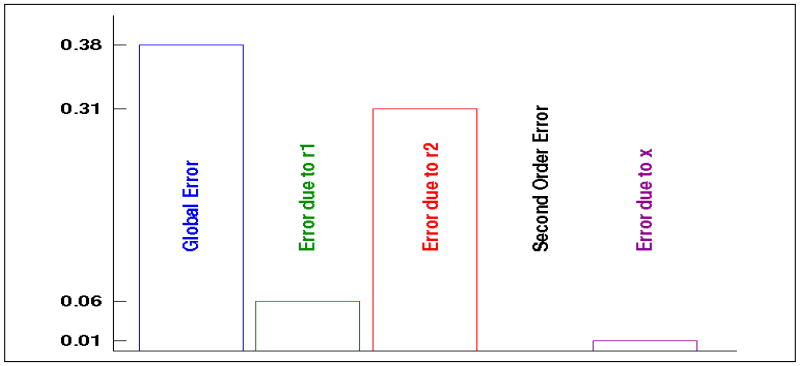


FIGURE 3 – Graphical representation of the errors defined in Equation (3).

The errors arising during the calculation of $a \times b$, which are given in Equation (3), can be represented by a graph which x-axis enumerates the control points of the program and which y-axis indicates the magnitude of the errors related to each point. The graph corresponding to Equation (3) is given in Figure 3. In general, Fluctuat displays such a graph for each variable visible at the end of the program being analysed (see Figure **??** in Section **??**).

More generally, the difference between the result x of a computation in real numbers, and the result f^x of the same computation using floating-point numbers, is expressed as

$$x = f^x + \sum_{\ell \in L \cup \{hi\}} \omega_\ell^x \vec{\varphi}_\ell \, . \tag{4}$$

In this relation, a term $\omega_\ell^x \vec{\varphi}_\ell$, $\ell \in L$ denotes the contribution to the global error of the first-order error introduced by the operation labelled ℓ. The value of the error $\omega_\ell^x \in \mathbb{R}$ expresses the rounding error committed at label ℓ, and its propagation during further computations on variable x. Variable $\vec{\varphi}_\ell$ is a formal variable, associated to point ℓ, and with value 1. Errors of order higher than one, coming from non-affine operations, are grouped in one term associated to special label hi, and we note $\mathcal{L} = L \cup \{hi\}$.

The result of an arithmetic operation $\Diamond$ at point ℓ_i contains the combination of existing errors on the operands, plus a new roundoff error term $\downarrow_\circ (f^x \Diamond f^y)\vec{\varphi}_\ell$. For addition and subtraction, the errors are added or subtracted componentwise :

$$x +^{\ell_i} y = \uparrow_\circ (f^x + f^y) + \sum_{\ell \in \mathcal{L}} (\omega_\ell^x + \omega_\ell^y)\vec{\varphi}_\ell + \downarrow_\circ (f^x + f^y)\vec{\varphi_{\ell_i}} \, .$$

The multiplication introduces higher order errors, we write :

$$x \times^{\ell_i} y = \uparrow_\circ (f^x f^y) + \sum_{\ell \in \mathcal{L}} (f^x \omega_\ell^y + f^y \omega_\ell^x)\, \vec{\varphi}_\ell + \sum_{\ell_1 \in \mathcal{L},\ \ell_2 \in \mathcal{L}} \omega_{\ell_1}^x \omega_{\ell_2}^y \, \vec{\varphi_{hi}} + \downarrow_\circ (f^x f^y)\vec{\varphi_{\ell_i}} \, .$$

Interpretation of other operation were also defined, that we do not detail here.

A natural abstraction of the coefficients in expression (4), is obtained using intervals. The machine number f^x is abstracted by an interval of floating-point numbers, each bound rounded to the nearest value in the type of variable x. The error terms $\omega_i^x \in \mathbb{R}$ are abstracted by intervals of higher-precision numbers, with outward rounding. This is implemented in the default analysis mode of Fluctuat, called "non relational abstract domain" at the graphic interface.

However, results with this abstraction suffer from the over-estimation problem of interval methods. If the arguments of an operation are correlated, the interval computed with interval arithmetic may be significantly wider than the actual range of the result. An implicitly relational version of this domain [18, 11] has been added more recently. Relying on affine arithmetic instead of interval arithmetic to compute the value f^x, it is in general a little more costly in memory and computation time, but more accurate. It is briefly described in Subsection **??**, and results with this analysis are shown in Section **??**. We also plan to add a mode that implements a relational domain for the errors.

Finally, an extension of the non-relational analysis has been added to analyse the amplitude and provenance of errors coming from a fixed-point implementation of a program. This implementation is given for the time being by assertions on the fixed-point format (word length, length of integer and fractional part) used for each float or double variable. A first prototype exists, of a tool that allows to determine automatically, from a floating-point program, fixed-point formats that would give the same order of accuracy for outputs of the program with the fixed-point semantics. This tool relies on analyses with Fluctuat with floating-point and fixed-point semantics, and on heuristics to progressively improve the fixed-point format.

5.3.2 Semantic of fixed-point numbers

The fixed-point mode allows the analysis of errors committed by using fixed-point numbers. Variables of the analysed program are still defined as double or float variables, but treated with a fixed-point behaviour, using assertions specifying the format of each variables, with notations much like those of System C. The parameters and assertions are respectively detailed in Section 2.4.3 and **??**.

5.4 Domains

5.5 Implementation of the analyser

Our prototype is based on a simple abstract interpreter developed at CEA. For the time being, if it can parse all ANSI C, but most library functions like the trigonometric functions are not implemented. The main objection to their implementation is that there are no clear specification for these functions. The analysis is inter-procedural, using simple static partitioning techniques.

The interval representing the floating-point value is implemented using the precision of classical floating-point numbers, and higher precision is used for the errors (necessary because numerical computations are done on these errors, and they must be done more accurately than usual floating-point). We use a library for multi-precision arithmetic, MPFR, based on GNU MP, but which provides the exact features of IEEE754, in particular the rounding modes. The interface is based on Trolltech's QT and communicates with the analyser through data files.

5.5.1 Implemented operations

The following operations are interpreted by the analyser :
- arithmetic operations : $+$, $-$, $*$, $/$; division and modulo for integer operands
- the square root `sqrt` and the absolute value `fabs`
- cast operations
- prefix and postfix increment and decrement operations : $i{+}{+}$, $i{-}{-}$, ${+}{+}i$, ${-}{-}i$; compound assignment operations : $+=$, etc ...

- logical operations, AND (&&), OR (||)
- conditions in expressions, eg $x = (y \leq z)$
- tests : $<$, $\leq$, $==$, etc ...
- bitwise operations : AND (&), exclusive OR ($\wedge$), OR (|), complement (~)
- bitwise shift operations : $<<$ and $>>$
- the `log` and `exp` operations are interpreted only for operands which floating-point value is a point (and not an interval). Indeed, the behaviour of these operations is not specified in the IEEE norm, and we can bound the error committed by using the floating-point operations only for a given input value. But perturbation errors of this input can be set, and a specification of bounds for values and errors for a specific implementation, if given, could easily be added.

5.5.2 Some specificities of the implementation

Integer computations (see Section ??) The modulo representation of integers is considered, and the user can specify on how many bits he wants every integer type to be represented. When a variable exceeds the maximum value of its type and loops to its minimum value (or the contrary), the difference between this value and the value that would be obtained if there was no limitation to integers, is taken as an error. Because of this modulo implementation, integer values are not always abstracted by an interval $[a, b]$, but also if necessary by a union $[\text{MIN}, b] \cup [a, \text{MAX}]$, where MIN and MAX are respectively the minimum and maximum possible values allowed by the type of the variable. Arithmetic operations and conversions are computed taking into account these two representations. This may sometimes lead to behaviours that are not obvious to understand, particularly in loops.

- bitwise operations : for these operations, we suppose that both the machine and the real value can be represented in the considered integer type. If necessary, when calling a bitwise operation, we reduce the errors on the operands so that the real value can not exceed the maximum integer of the type.
- conversion floating-point number $\rightarrow$ integer, division on integers, modulo and all bitwise operations : in order not to lose too much precision, the list of errors of the operand is transmitted on the result in only one error interval corresponding to the point of the operation (the provenance of errors is lost in these cases)

Floating-point computations

- unspecified behaviour (see Section 4.4.3)
  - order of evaluation in arithmetic expressions : the computation of the super-set of possible floating-point values relies on the assumption that we know the order of evaluation of arithmetic expressions (the more natural, left to right, evaluation), and that the compiler makes no reorganisation. Unfortunately, this is frequently not the case, and the compiler reorganises computations even in spite of parentheses. The most secure way to prevent this would be to split computations using intermediary variables, for example using an appropriate pre-treatment.
  - storage of intermediary results in internal registers, with a precision different from the IEEE floating-point : we have noted such behaviour, which results in that the values given by the analysis and the execution are different. Using new variables to assign intermediary computations in the source code should solve this problem.
- the `fabs` function applies to a double precision operand, and its result is a double precision number. Therefore, in examples when the operand is an interval not reduced to one value, even if we know that it is a simple precision number, a rounding error will be attributed to the computation.

Tests Possibly unstable tests (when the machine and real values may not follow the same path in a test) are signalled by a message on the command line and in a file `unstable_test.out` in the `Fluctuat_ProjectName` directory, giving the line number in the source code where the test occurs. If there is no unstable test, this is also specified in the `unstable_test.out` file. For the time being, the errors are computed with the hypothesis that the execution of the program with real number would follow the same path as the execution with finite precision numbers. This can lead to false estimations of some errors.

Execution errors We suppose that execution errors cannot occur. In case the analyser detects a possible error, it may signal it by a message on the command line but goes on with the analysis.

5.5.3 Known limitations and bugs

Loops with composed entering/exit condition The analysis on some examples with a loop which entering/exit condition is a composed condition, may not terminate. This happens in some cases when this condition includes the operator ||, for example in (see EXAMPLES/boucle_doubleconditions_2/boucle_doubleconditions_2.c)

```
int i = 0;
while ((i<=10) || (i<=10000))
  i++;
```

A solution in some cases is to reverse the order of the two conditions, writing here

```
int i = 0;
while ((i<=10000) || (i<=10))
  i++;
```

But most of the time, these forms are a problem for the analyser.

Structs and arrays The arrays can be used as parameters of a function, but only as explicit pointers. For example, the following prototype of function f

```
voif f(int Tab[])
```

has to be rewritten as

```
void f(int *Tab)
```

Pre-processor instructions Some pre-processor directive are ignored by Fluctuat :

- ♯pragma.

A Installation of the tool

To install Fluctuat, the first step is to uncompress the archive with :

```
tar xvzf fluctuat.tgz
```

The distribution contains some directories under the main one :
- `ABSINT` with included files used by the analyser,
- `bin` for the binaries (fluctuat, daed_analyze, daed_project),
- `DEMO` with a set of examples,
- `MANUAL` contains the electronic and html versions of this manual.
- `scripts` for the pre-processing scripts.

Two dependencies are needed in order to use Fluctuat :
- `libqt` Qt graphical framework v3 libraries
- `gcc` GNU C Compiler

You also have to set `FLUCTUATHOME` environment variable to the root directory of the location at which you have installed the distribution. And we recommend to set your `PATH` variable with the following command :

```
export FLUCTUATHOME=_path_to_fluctuat_distribution_
export PATH=$FLUCTUATHOME/bin:$PATH
```

Once it is done, you only have to type `fluctuat` in a terminal to start the tool. Thanks to previous commands, you can also use Fluctuat in batch mode with the `daed_project` and `daed_analyze` commands.

B Actions reference card

Menu	Details	Section	Shortcut
▷ File			
Create project	Launch the project creation	2.3.1	Ctrl+N
Open project	Open an already created project	2.3.2	Ctrl+O
Edit source	Edit project source files	2.3.3	Ctrl+E
Exit	Quit Fluctuat	–	–
▷ Analysis			
Start analysis	Select the parameters and launch the analysis	2.4.1	Ctrl+A
Quick analysis	Launch the analysis	"	Ctrl+Shift+A
Start interactive analysis	Launch the interactive analysis	2.7	Ctrl+D
Abort analysis	Stop the current analysis	"	–
View parameters	View the last used parameters	2.4.2	–
View log	Display the analyzer output	2.5.1	Ctrl+L
View error log	Display the analyzer error output	"	Ctrl+Shift+L
View monitor	During an analysis, display the progress	2.4.4	Ctrl+M
View statistics	Show full statistics about last analysis	2.5.1	–
▷ Results			
View in/out variables	Show the in/out variables	2.5.5	–
View dead code	Display the dead code in the source code window	2.5.6	–
View warnings	Show warnings of the analysed program	2.5.4	–
Dump results	Save analysis results	2.5.7	–
Worst case	Generate worst reachable case	2.5.10	–
Sensitivity analysis	Show sensitivity of the input variables	2.5.11	–
▷ Error-mode			
Absolute error	Display absolute value in the origins of errors	2.5.2	–
Interval mode	Display interval value in the origins of errors	"	–
Biggest error	Go to the biggest contribution in the origins of errors	"	–
Next error	Go to the next contribution	"	–
View float evolution	Display evolution of the value of the selected variable	2.5.3	–
View real value evolution	Display evolution of the real value	"	–
View error evolution	Display evolution of the error	"	–
View relative error evolution	Display evolution of the relative error	"	–
▷ Font			
Small font	Display the source code in small font	2.2	–
Normal font	Display the source code in default font	"	–
Big font	Display the source code in big font	"	–
▷ ?			
User manual	Show this manual	–	–
About Fluctuat	Show license and legal information	2.8	–

C Interactive analysis reference card

FLUCTUAT Interactive Analysis

Essential Commands

Ctrl-D	show the window of the interactive analysis you can now enter commands
b [file:line]	set breakpoint in file at line
r function	start the analysis of function
p var	print the range and the error of var
view var	display the range and the origin of errors for var
c	continue analyzing your program
n	next different line, stepping over function calls
s	next different line, stepping into function calls

Starting and Stopping the Analysis

Ctrl-D	show the window of the interactive analysis you can now enter commands
setr file	load a resource file (eg: project.rc) (see setp to change an analysis parameter)
Ctrl-G	interrupt the analysis (see info count ⇒ absolute progression number)
Ctrl-K	kill the interactive analysis (see Ctrl-D to start a new session)

Analysis Control

c [count]	continue the analysis; if count specified, ignore
continue	this breakpoint next count times
s [count]	continue until another line is reached
stepi [cnt]	step by internal control flow graph instructions (see info count ⇒ absolute progression number)
n [count]	continue until another line is reached (in the same function or in a calling function)
finish	continue until the current function returns
set var **in** [min, max]	
	assume var is in [min, max] (~ DBETWEEN)
seta var **in** [min, max], [exactmin, exactmax], [errmin, errmax]	
	assume var is in … (~ DREAL_WITH_ERROR)
union var **with** [min, max]	
	add [min, max] to the values that var may take
setr file	load a resource file (eg: project.rc)
setp parameter value	eg: setp unfoldloopnumber 500
	set the FLUCTUAT parameter to value

Breakpoints

break [file:line]	set breakpoint in file at line
b [file:line]	eg: break main.c:24
break fun	set breakpoint at function “fun”
b … **if** var **is top**	break if var becomes top
b … **if** var **is bot**	break if var becomes bot
b … **if** var = float	break if float is a potential value for var
b … **if** var **is in** [min, max]	
	break if var may have a value in [min, max]
b … **iferr** var **is top**, **is bot**, = float, **is in** [min, max]	
	break if var may have an error in …
b … **ifexp** exp **is top**, **is bot**, = float, **is in** [min, max]	
b … **iferrexp** exp **is top**, **is bot**, = float, **is in** [min, max]	
info break	show defined breakpoints
del [n]	delete breakpoints
dis [n]	disable breakpoints
en [n]	enable breakpoints
ign n count	ignore breakpoint n count times
commands [n]	execute command-list every time
command-list	breakpoint n is reached
end	

Display

print var	print the range and the error of var (also **p**)
affprint var	print the range and the origins of the errors for var
view var	display the range and the origin of errors for var
draw x,y(,z)	draw the 2 (3) dimensional zonotope (also **drawexp**)
pexp expr	print the range and the error of expr expr is parsed by the parser in a recovered scope
vexp expr	display the range and the origin of errors for expr
info count	return the absolute number of instructions abstractly interpreted by the analysis (see **stepi**)
getp parameter	
	return the value of the FLUCTUAT parameter eg: getp unfoldloopnumber

Journal

Ctrl-R	replay the commands in the journal except the last one
replay [n]	replay the n first commands stored in the journal

D Simple to advanced parameters

Simple parameters are a subset of advanced parameters, the following tables make the links from simple to advanced parameters :

Simple parameter	Advanced parameter	Values
▷ General :		
Analysis type	Abstract semantics	Abstract simulation → Symbolic execution
Analysis mode	Abstract semantics	4 values from "Quick" to "Accurate" 1^{st} → Non-relational 2^{nd} → Relational on values, 3^{rd} → Global error, 4^{th} → Fully relational.
Start function	Entry point	-
▷ Analysis accuracy :		
Internal accuracy	MPFR bits	5 values from "Low" to "Full" 1^{st} → 60, 2^{sd} → 100, 3^{rd} → 500, 4^{th} → 1000, 5^{th} → 5000.
Subdivides bitwise operations	Refinement of bitwise operations	Unchecked → 0 Checked → 100000
▷ Loops :		
Initial unfolding	Initial unfolding	5 values from "Simple" to "Full" 1^{st} → 10, 2^{nd} → 50, 3^{rd} → 100, 4^{th} → 200, 5^{th} → 500.
Unfolding	(Widening Threshold,Cyclic unfolding)	8 values from "Simple" to "Full" 1^{st} → (5,1), 2^{nd} → (50,5), 3^{rd} → (100,5), 4^{th} → (50,10), 5^{th} → (100,10), 6^{th} → (50,50), 7^{th} → (100,50), 8^{th} → (200,50).
▷ Subdivisions :		
Subdivision 1 and 2	Subdivision 1 and 2	-
Use parallel subdivisions	Number of parallel threads	Unchecked → 0 Checked → 4
▷ Environment :		
Execution norm	(Size of "float" mantissa, Size of "double" mantissa, Rounding mode)	IEEE754 → (24,53, Nearest) MIL1750 → (24,40, Toward minus infinity)

E Parameters

All parameters and options of the analysis accessible through the user interface can be set in the resource file, using keywords. Note that not all of these parameters have to be set to a value, there are default values for all of them. Also, the resource files created by the user interface (named 'project.rc') can be edited and used for further analyzes, but these must be done in batch mode (using the interface erases the old resource file). Each line begins with a '*', then a keyword and its value preceded by '='. A very simple resource file :

```
* analysyskind = 0;
* entryname = main;
* initvar = 0;
```

We describe, for each option in the user interface, the corresponding keyword and if necessary its use. If it is not specified, for boolean parameters, the possible values are 0 for false and 1 for true, and for other parameters, it takes a numerical value.

- `Abstract semantics` → `analysiskind` :
  - 0 for non-relational analysis,
  - 1 for weakly-relational analysis,
  - 2 for fixed-point analysis,
  - 4 for fully-relational analysis,
  - 5 for fully-relation (global error) analysis.
- `Analysis type` → `execution`
  - 0 for static analysis,
  - 1 for symbolic execution.
- `Entry Name` → `entryname`
  the name of the start function.
- `Interpret float->integer casts as errors` → `intfix`
- `Test interpretation on float only` → `floatflow`
- `Number of parallel threads` → `threadsize`
- `MPFR Bits` → `mpfrbits`
- `Noise symbol creation constraint` → `noisecreation`
- `Initialise variables to 0` → `initvar`
  - 0 for an initialisation to 0,
  - 1 for an initialisation to top.
- `In/out analysis` → `inout`
- `Storage` → `storage`
- `Initial Unfolding` → `nbexecbeforejoin`
- `Depth of inter-loop relational history` → `nbrelationlevel`
- `Cyclic unfolding` → `unfoldloopnumber`
- `Keep relations at the end of loops` → `keeprelations`
- `Unfold non-relational errors` → `errorunfold`
- `Global union` → `globalunion`
- `Automatic Parameters` → `iterauto`
- `Maximum Iterations` → `maxiter`
- `Widening Threshold` → `wideningthreshold`
- `Narrowing Threshold` → `narrowingthreshold`
- `Use progressive widening` → `progressivewidening`
- `Loss of precision (bits)` → `reductionstep`
- `Minimal precision` → `minprec`
- `Threshold before standard widening` → `infthreshold`
- `Exact error calculation for constants` → `calculexacterr`
- `Convergence of higher order errors` → `relaxcvordresup`
- `Use subdivisions for the whole program` → `subdivide`

- File of first subdivision → anasubfile
- Line of first subdivision → linenumber
- Number of slices for first subdivision → slices
- File of second subdivision → anasubfile2
- Line of second subdivision → linenumber2
- Number of slices for second subdivision → slices2
- Refinement of bitwise operations → nbdivisions
- Size of ‘‘char’’ type → MAX_CHAR, MIN_CHAR, MAX_SCHAR, MIN_SCHAR and MAX_CHAR.
  For each resource name (“char”, “short char” and “unsigned char”), you have to set the maximal and minimal values.
- Size of ‘‘short’’ type → MAX_SHRT, MIN_SHRT and MAX_USHRT.
- Size of ‘‘int’’ type → MAX_INT, MIN_INT and MAX_UINT.
- Size of ‘‘float’’ mantissa → fmantissa
- Size of ‘‘double’’ mantissa → dmantissa
- Rounding mode → roundmode
- Levels of abstraction → alias
  - 0 for No pointers,
  - 1 for Abstract arrays as one location,
  - 2 for Unravel arrays.
- Do not limit call stack → nolimitstack
- Call Stack Depth → callstackdepth
- Overflowmode → fixoverflowmode
  - 0 for Wrap around,
  - 1 for Saturate.
- Rounding mode → fixroundingmode
  - 0 for Truncate,
  - 1 for To the nearest.
- Machine word length → fixmwm

Références

[1] Polymake wiki. http://http://www.polymake.org/doku.php.

[2] Mil-std-1750a - military standard sixteen-bit computer instruction set architecture. Technical report, 1982.

[3] Ieee 754 standard for floating-point arithmetic. Technical report, 2004.

[4] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *SIBGRAPI'93, Recife, PE (Brazil), October 20-22*, 1993.

[5] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Symbolic Computation*, 2(4) :511–547, 1992.

[6] M. Daumas and J. M. Muller, editors. *Qualité des Calculs sur Ordinateur*. Masson, 1997.

[7] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1), 1991.

[8] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[9] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : a simple abstract interpreter. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[10] E. Goubault, M. Martel, and S. Putot. Concrete and abstract semantics of floating-point operations. Technical Report DRT/LIST/DTSI/SLA/LSL/01-058, CEA, 2002.

[11] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS'06, Seoul, August 29-31*, 2006.

[12] G. Hanrot, V. Lefevre, Rouillier F., and P. Zimmermann. The MPFR library. Institut de Recherche en Informatique et Automatique, 2001.

[13] W. Kahan. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Technical report, Berkeley University, 1996.

[14] M. Martel. Propagation of roundoff errors in finite precision computations : a semantics approach. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[15] M. Martel. Static analyses of the numerical stability of loops. In *Static Analysis Symposium, SAS'02*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[16] M. Martel. An overview of semantics for the validation of numerical programs. In *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, 2005.

[17] D. Monniaux. The pitfalls of verifying floating-point computations. Technical report, CNRS / Ecole normale supérieure, 2008.

[18] S. Putot and E. Goubault. Weakly relational domains for floating-point computation, 2005.

[19] S. Putot, E. Goubault, and M. Martel. Static analysis based validation of floating point computations. In *Novel Approaches to Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 306–313, Dagstuhl, Germany, 2004.

[20] J. Stolfi and L. H. de Figueiredo. An introduction to affine arithmetic. *TEMA Tend. Mat. Apl. Comput.*, 4(3) :297–312, 2003.